Ph.D. DISSERTATION

Efficient and Practical Security Monitoring on System-on-Chip with Application-Specific Hardware Modules

시스템온칩 상에서의 효율적이고 실용적인 보안 모니터링을 위한 응용 특화 하드웨어 모듈

BY

Ingoo Heo

FEBRUARY 2016

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Ph.D. DISSERTATION

Efficient and Practical Security Monitoring on System-on-Chip with Application-Specific Hardware Modules

시스템온칩 상에서의 효율적이고 실용적인 보안 모니터링을 위한 응용 특화 하드웨어 모듈

BY

Ingoo Heo

FEBRUARY 2016

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Efficient and Practical Security Monitoring on System-on-Chip with Application-Specific Hardware Modules

시스템온칩 상에서의 효율적이고 실용적인 보안 모니터링을 위한 응용 특화 하드웨어 모듈

지도교수 백윤흥

이 논문을 공학박사 학위논문으로 제출함

2015 년 12 월

서울대학교 대학원

전기 컴퓨터 공학부

허인구

허인구의 공학박사 학위논문을 인준함

2016 년 1 월

위 원 장	이혁재	(인)
부위원장	백윤흥	(인)
위 원	김태환	(인)
위 원	강병훈	(인)
위 원	신준범	(인)

Abstract

Many researchers have proposed the concept of security monitoring, which watches the execution behavior of a program (e.g, control-flow or data-flow) running on the machine to find the existence of malicious attacks. Among the proposed approaches in the literature, software-based works are known to be relatively easy to be adopted to the commercial products, but may incur tremendous runtime overhead. Although many hardware-based solutions provide high performance, the inherent problem of them is that they usually mandate drastic change to the internal processor architecture. More recent ones to minimize the change have proposed external devices for security monitoring. However, these approaches intrinsically suffer from the high overhead to communicate with their external devices. Consequently, they either significantly lose performance, or inevitably make invasive modifications to the processor inside.

In this thesis, I propose several approaches for efficient security monitoring, where external hardware engines conduct the task of monitoring. The main priority in desinging the engines is not to require any modification in the host processor core internal. Thus, the engines introduced in this thesis are designed as external hardware modules and integrated to the host processor using the existing interface in the system. Complying with the rule, I explored the architectural design space for the engine and in ths thesis, three types of such approaches will be presented. Starting from the hardware engine that utilizes only the system bus, I will introduce the final solution that exploits the debug interface of the commercial processor. From the design exploration, this thesis shows various design decisions that can be applied in the current commercial

platforms.

Keywords: Security monitoring, debug interface, SoC level integration, external hardware engine

Student Number: 2009-20920

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Nowadays, many types of attacks threaten the security of computer systems such as desktop machines, mobile phones, tablets and even internet-of-things (IoT) devices. In general, adversaries try to find the vulnerability of the victim system and exploit it to launch an attack, thereby achieving their malicious goal like leaking information, compromising the system or performing denial-of-service attacks [12]. Once the attack is successfully executed, the target machine can be damaged or controlled by the adversary at her disposal.

In order to protect the systems from the attacks, various solutions have been proposed, such as cryptography [83], execution environment isolation [13], randomization [89] and system monitoring [1, 62]. Among them, the system monitoring, which watches the execution behavior of the program running on the machine to find the existence of attacks, is one of the most popular and widely used ways. In this approach, they firstly define a set of legitimate rules that should comply with during normal code execution. Then they check at runtime if there is any violation of the rule, which can be regarded as the

symptom of an attack. Many monitoring methods proposed recently, such as dynamic information flow tracking (DIFT) [62], memory bound checking [22] and control flow integrity (CFI) checking [1], have evinced that they are effective in detecting various harmful attacks and keeping systems secure.

These security monitoring schemes can be implemented in various forms of either software or hardware. In most software approaches, they add instrumented code into the original application to perform the proposed monitoring mechanism [1,17,62]. Their key advantage is that they can perform the security monitoring simply by programming their algorithms on the existing hardware architecture. Not surprisingly, however, they show too large computing overhead to be deployed in practice. For example, in the case of DIFT, even after much effort [20, 73], the overhead still remains one or two orders of magnitude higher than that of hardware approaches [23, 87] in which extra hardware for monitoring operations is designed and integrated into an existing processor for acceleration. The hardware typically consists of logic blocks that observe the execution of each instruction in the processor and keep track of information flowing from the execution unit at every cycle, in order to monitor the execution behavior of the program [45, 46].

Unfortunately, the remarkable speed of hardware-based monitoring comes at a cost. To maximize the performance, the hardware has been tightly integrated inside the processor. However, such integration mandates major modifications to processor internal components such as registers and pipeline datapaths, thus substantially increasing the time and cost for re-manufacturing existing processor core architecture [44]. As alternatives to mitigate this problem, there have been more recent studies [19, 44, 59] that propose the techniques aiming to minimize the change to the processor core internal. In their approaches, the host processor can concentrate on the execution of its own code while the

time-consuming monitoring task is offloaded to the specialized hardware module outside the processor. In the literature, they empirically demonstrated that their monitoring scheme can be carried out in a great speed by external hardware, relieving significant burden for the extra computation from the host.

Nevertheless, there still remains a great challenge to overcome for the success of these approaches. The challenge originates from the limited ability of an external module to watch every internal state change dynamically made by the code running on the host. For precise security monitoring, the external monitor should be able to receive from the host virtually all essential runtime information such as branch targets, memory addresses and register moves, which will incur a tremendous amount of traffic for communication between the two devices. In [19,59], they report that the communication overhead may account for up to 30% of the total execution time even after all their optimizations through hardware communication buffers and special instructions. In [44], this overhead issue was treated more aggressively by modifying the host architecture in a way that a customized interface for the communication can be embedded into the processor pipelines. Through this interface, their external device was able to have a special connection with which any runtime information for monitoring can be extracted directly from the internal pipeline with very little overhead. However, the main drawback is that the host processor should be still modified for the customized connection, as in the previous hardware approaches [23,87].

In this thesis, I will discuss several studies on the external hardware engines for security monitoring techniques. The goal of this thesis is to explore the architecture design space for the hardware monitors and finally propose a viable solution that overcomes the limitations of the previous works. The foremost priority in design is that the original architecture of the host processor is not modified and the hardwre monitor should be attachted in the system via exist-

ing interfaces such as the system bus. Complying with these rules, we explore various architectural choices for the monitors.

First of all, in Chapter 2, I will introduce the external hardware engine for the security monitoring techniques based on tag processing. The main purpose of this engine is to provide a flexibility so that various monitoring methods can be performed on the same monitor architecture. For this, we design the engine as an application specific instruction processor which have a set of instructions specialized for the monitoring tasks.

In Chapter 3, I will explain another hardware engine, which exploits an on-chip debug module embedded in the ARM processor architecture. The original purpose of the debug module is to provide various runtime information for debugging to the external hardware debugger without affecting the host performance. In Chapter 3, I integrate the hardware engine to the debug module so that such information can be delivered to the engine. By doing so, the performance overhead required for delivering the runtime information can be eliminated.

In Chapter 4, I will introduce my final solution. The key idea is to connect the hardware engine to the core debug interface in a processor, instead of the on-chip debug module. In general, the on-chip debug module of a processor is connected to the core debug interface which extracts the runtime status of the host from the processor internal pipeline. Since the core debug interface provides more information than the on-chip debug module, the hardware engine can avoid the lack of information and does not require the meta-data. In the experiments, it is shown that this approach can reduce the performance of security monitoring substantially, while the required hardware resources and memory space overheads are also reasonable.

This thesis is organized as follows. From Chapter 2 to 4, I will present the

hardware engines discussed above. For each hardware engine, I will discuss the contribution of the design and the related prior works. Also, each corresponding chapter includes the detailed architecture of the engine, the implementation details and the experimental results. After the final solution is introduced in Chapter 4, I will conclude this thesis in Chapter 5.

Chapter 2

Implementing an Application Specific Instruction-set Processor for System Level Dynamic Program Analysis Engines

2.1 Introduction

Dynamic program analysis (DPA) is to analyze software code as it executes on a processor. In recent years, it has been widely used in profiling system performance, finding software bugs for reliability and runtime monitoring for system security. As an example, Memcheck [82] is a DPA tool implemented in the Valgrind binary instrumentation framework [60] and uses dataflow tracking to observe the memory usage behaviors of the target applications to detect unintended misuses of memory. Dynamic information flow tracking (DIFT) is also a representative DPA technique which tracks and restricts the use of designated data by managing metadata called *tag*. In many studies, DIFT has been used to effectively resolve their various problems such as runtime mon-

itoring [23, 62] or malware analysis [11]. Likewise, DPA has been applied to enable many other types of techniques [27] such as memory protection [96], array bound checking [29], software debugging support [60] and garbage collection [43]. Consequently, with the ever-increasing importance, the use of DPA is being expanded to a wide range of security and reliability problems.

To achieve their goals of DPA, many researchers rely on dynamic binary instrumentation (DBI) frameworks such as Valgrind [60], Pin [53], and DynamicRIO [14]. While dynamic analysis through software DBI provides complete analysis environment with the extreme flexibility, the amount of analysis at either test-time or runtime is bounded by the performance impact that can be tolerated [91]. The performance overhead is especially crucial in complex DPA techniques which require amount of computation as the target program executes. For example, LIFT [73], a DIFT solution with DBI tool, slows down the program execution by around 4 times at runtime even with aggressive optimizations. Although several approaches have been proposed to utilize multiprocessors [19, 59, 63] that are readily available in modern multicore architectures where each core is a general-purpose processor (GPP), they could also not achieve sufficient performance improvement mainly because the original architectures were not optimized for DPA in the first place [91].

To address the shortcoming of software-based analysis, several *core-level* hardware supports for DPA have been proposed [19, 23, 27, 28, 87, 92], where extra hardware logic customized for DPA operations is integrated into a processor core. Even though they could bring the overhead down to a few percents, they require invasive modifications to the core internal (e.g., registers and pipeline data paths). In fact, microprocessor development may take several years and hundreds of engineers from an initial design to production [44]. Therefore, the substantial costs of development to integrate the logic would hamper proces-

sor vendors to adopt new hardware unless its generality and versatility are clearly proven. For this reason, some proposed a flexible core-level accelerator integrated in a processor that can support a set of diverse DPA functions by reconfiguring the accelerator [27]. However, they still have a limited functional extensibility in that a new DPA function cannot be supported by hardware unless it was considered in the initial hardware design.

As an alternative direction to avoid invasive core-level modification, a *system-level* DPA acceleration engine was proposed, which is integrated into a system by being connected to the processor through existing channels such as peripheral interfaces. In [91], they built a working prototype, called *Hardgrind*, where the engine is implemented as an external device and connected to the host system via a PCI bus. In the experiment, they demonstrated that even without internal changes to an existing CPU, heavy-weight DPA tools [82] benefit from the acceleration strategy, and the speedup can be great, being up to 4.4 times faster than pure software techniques stated above. These results reveal a potential advantage of a system-level DPA engine that it may offer a more affordable solution to extend the engine for new DPA functions than the core-level ones because the extension could be made separately from the host system without necessitating the overall host architecture modification. Such an advantage would be particularly beneficial to recent mobile SoC platforms where the system-level integration provides a better extendibility by enabling the *platform-based design* which is a de facto standard methodology to develop complex SoCs including commercial products like smartphone application processors (APs). Because the platform-based design tends to foster systematic reuse of already-implemented modules [10], it is important to preserve the other components intact when a functionality like DPA is additionally supported. In the system-level approach, all special logics customized for DPA are fully in-

tegrated into an independent module so that the other modules can be reused thereby lowering development risks, costs and time to market. Although the existing system-level approach [91] has evinced not only a great potential in performance but also the extendibility to support a variety of DPA methods, they have not described the detailed architecture of their hardware engine or its implementation. Instead, by assuming it as simple core logics for analysis methods [90, 103], they have just tried to quantify the potential of their approach. Therefore, in order to leverage the deployment of the system-level engine in real machines, it is mandatory to consider the realistic design issues as the engine being implemented for the component in an existing SoC.

For this purpose, in this chapter, we propose a DPA hardware engine, called the *program analysis unit* (PAU), which has been fully implemented and integrated as a system level component in an existing computing platform. The novelty of our engine is that it is software programmable in order to attain not only the high performance but also the great expandability of our DPA solutions. For this, we have implemented PAU in the form of an application specific instruction-set processor (ASIP) whose instruction-set is customized to reflect common features of various DPA methods. First, by enabling the user to decouple DPA operations from the host code and accelerate them on PAU, we have substantially reduced the performance overhead of DPA. Furthermore, in practice, PAU can execute any designated DPA as software codes running on the processor, offering a great deal of flexibility and extensibility for a wide range of DPA functions.

To examine the effectiveness of our approach, we chose three exemplary DPA techniques for case studies: DIFT, Uninitialized Memory Checking (UMC) and Bound Checking (BC). We implemented the DPA schemes with the software code for PAU and enabled it to carry out the DPA computations off-loaded

from the host CPU. Also, we built an in-house instrument tool to insert data gathering code segments for the CPU and generate actual analysis codes for PAU automatically. By mapping those codes to both processors, we have parallelized DPA computations between the CPU and PAU thereby improving the analysis performance. In addition, our DPA engine was able to adopt the optimization techniques suggested in the software-based approaches [22, 73, 93, 104] simply by programming them on PAU. The case studies show that our approach can be applied to various time-consuming DPA techniques by providing hardware-backed power in performance as well as software-based flexibility in analysis.

In order to show our experimental results on a working prototype SoC, we implemented our proposed design in RTL and the full system is prototyped on a Xilinx Virtex-5 FPGA board. Recent experiments have demonstrated that our proposed design can enhance the performance for the three implementation examples substantially as compared when the DPA schemes are conducted by pure software-based approaches. Furthermore, our PAU is far more energy/area efficient than general purpose commodity cores.

In this chapter, we make the following contributions:

- We proposed PAU, which is a system-level hardware DPA engine that does not require the modification of the host core. We designed PAU as an ASIP to achieve both the programmability and the acceleration of hardware.
- We implemented our PAU with Verilog HDL and integrated it into a SoC prototype to build a full-system. We, then, measured the overheads of PAU in terms of performance, area and power by running mibench [40] benchmark to empirically show the efficacy of our approach.
- To show the programmability of PAU, we chose three well-known DPA

techniques (i.e., DIFT, UMC and BC) and implemented them on our PAU.

This chapter is organized as follows. Section 2 explains the background of tag-based DPA techniques and how a system-level hardware helps this DPA execution. Section 3 gives an architectural/functional overview of our ASIP, and Section 4 describes the programmable processing core of the hardware engine. After our case studies are introduced in Section 5, Section 6 will discuss how software optimizations for DIFT can be adopted into our PAU with the help of its flexibility. Then, Section 7 reports the experimental results and Section 8 relates our work with others. Finally, in Section 9, we will summarize this chaper.

2.2 Backgrounds

To enable our PAU to cover the broad class of DPA, we should look into several widely-used DPA techniques and figure out the characteristics that are commonly inherent in those. For this reason, in this section, we will first introduce the generalized DPA model which has been proposed in previous literature [19, 27], in order to understand the commonalities of DPA. Then, we will explain the execution flow of DPA with a system-level hardware engine.

2.2.1 Understanding Tag-based DPA Techniques

To understand the core features of various DPA techniques, it is noteworthy that previous studies [18, 19, 27] have already analyzed a number of the techniques whose characteristics [22, 29, 62, 93, 103] are summarized in Table 2.1. Note in the table that many techniques commonly maintain and check the meta-data information, called *tag*, to describe the status of the host program [27] despite the differences in their types or granularities. For example, DIFT maintains

a 1-bit tag to indicate whether a word/byte is from a potentially malicious sources [62]. On the other hand, a tag may be associated with a location such as a memory address instead of a value to keep information on the properties of storage itself, as in memory bound checking [29]. Also, some of DPA schemes keep coarse-grained tags for composite program objects such as records and arrays [43].

With the tags, DPA generally conducts mainly two types of tag operations to achieve its purposes; tag updating and tag checking. Throughout program execution, DPA maintains the tags by updating the tags at specific events. Some tags are occasionally updated at certain events such as library calls while the others might be updated at nearly every monitored instruction. Tag checking is to test if an invariant of the program is violated. In DIFT, for instance, an alarm is triggered when any of the data from untrusted sources involve in potentially illegal activities by checking the tag. With these two types of tag operations, DPA can find bugs, profile system performance or detect various attacks on monitored programs. In this chapter, we have designed our PAU based on the *tag-based DPA model*.

2.2.2 DPA Execution on a System-Level Hardware Engine

This subsection describes the DPA execution flow in the system-level hardware approach, where the system mainly consists of a host CPU and an off-core hardware engine, as depicted in Figure 2.1. In the system, the host is regarded as a producer that gathers the data required for analysis, called the *execution traces*, then sends them to the analysis hardware engine. Conversely, the off-core engine is regarded as a consumer that receives the traces and performs the actual analysis task. For example, in case of Memcheck [82], the host captures memory access behaviors of the monitored program, such as accessed memory addresses

DPA Technique	Tag Type	Description	Tag Operations
DIFT	1-bit tag to indicate taintness	Prevents common malwares from leaking the critical information by tracking and limiting uses of untrusted I/O inputs.	tag update tag checking
Uninitialized Memory Checking	1-bit tag to present whether the memory location has been initialized	Checks whether a certain memory location is initialized before reading it.	tag update tag checking
Bound Checking	multi-bit tag to match the data and its location	Both tags for memory location and tags for pointers are set. On each memory access instruction, the tag of the pointer that is used to access memory is compared to the tag of the accessed memory location.	tag update tag checking
Reference Counter	multi-bit tag to store referenc e count for the data location	Performs reference counting to aid garbage collection mechanism. On an instruction that creates a new pointer, the tag is incremented. On an instruction that destroys an existing pointer, the tag is decremented.	tag update tag checking
LockSET	1) multi-bit tag for the current set of locks held by the thread 2) multi-bit tag to maintain a candidate set of locks	Performs race detection among multithreaded applications.	tag update tag checking

Table 2.1 Tag types and operations of several DPA schemes

and a set of data written to the memory. Then, the captured information is transferred to the analysis engine which performs the analysis task that tracks dataflow and detects unintended memory uses in the program by updating and checking the tags. As presented in much literature [19, 27, 28, 44, 91], these approaches with separate engines have shown to be efficient because it relieves the burden of the host CPU by reducing the competition for resources (i.e., cycles, registers and caches) between the original program and DPA.

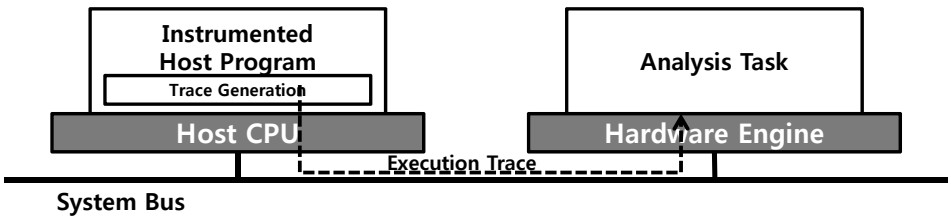


Figure 2.1 Execution model of system level hardware engine

As explained, in this execution model, the execution traces should be created

and then transferred to the analysis hardware. In the core-level approaches [18, 19, 27, 44], the traces are transparently gathered with a dedicated hardware that observes the instructions executed by the monitored program and creates the corresponding execution trace. On the other hand, in the system-level approach [91], the host program is augmented with the code for trace generation so that a stream of traces is created by the code on the host. Meanwhile, the analysis task on the hardware engine can be implemented in the form of either hardware or software. In Hardgrind, the analysis tasks of MemCheck [82] and Helgrind [79] are implemented with specialized hardware modules like Range Cache [90]. With the help of ASIC-style design, Hardgrind could achieve the speedups from 29% to 440% in the two DPA techniques.

2.3 System-Level Programmable DPA Engine for Extendibility

In this section, we will give an architectural overview including our hardware engine and discuss how DPA is performed on the proposed system. Also, we will discuss the efficient communication strategy between the host and our engine.

2.3.1 Overall System Design with PAU

Based on the execution model of the system-level approach introduced in the previous section, we designed our overall system which mainly consists of a host CPU and PAU as depicted in Figure 2.2 where PAU is connected via a general system bus to the host CPU along with other modules including special purpose processors.In this work, our SoC employs an AMBA-compliant system bus [51] which is a shared bus architecture conforming to the AMBA protocol, a de-facto standard for master-slave communication in modern SoC design. Hence,

as long as our design obeys the AMBA protocol, it can be used in every SoC based on AMBA protocol without any hardware change.

The key components of PAU are the *tag processing core* (TPC) and the *main controller*. TPC is a processor core of PAU which executes software codes. Its main task is to perform tag operations along the program execution flow running on the host and analyze the monitored program. We will postpone the discussion of this task to Section 4. The main controller manages all transactions related to the DPA computation. It contains *configuration registers* whose values can be changed to specify various types of transactions. Thus, the host can control the action of PAU directly by setting these registers to certain values. To facilitate this control from the host, the configuration registers are memory mapped.

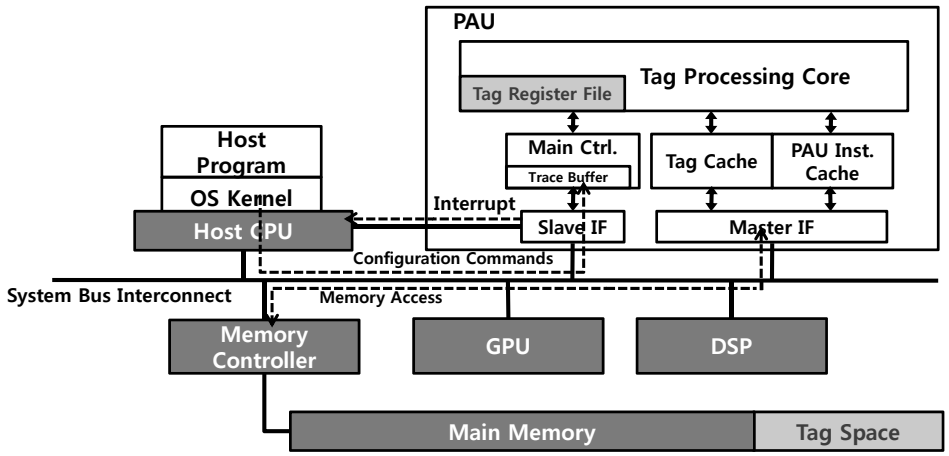


Figure 2.2 The overall system design with PAU

The central role of PAU is the management of all the tags used for DPA. During DPA computation, all the tags being accessed are located in either PAU or the main memory. For every host processor register, TPC has a corresponding 32-bit tag, all of which together form a single register file in TPC, called the

tag register file (TRF). Since our host processor has 32 general registers, the TRF also consists of 32 entries. Since many DPA schemes augment tags to the registers, it is efficient to employ the TRF to support the tag-based DPA. Likewise, we allocate a space in the main memory, called *tag space* [92], to manage various types of tags in the memory. This space is maintained by TPC throughout program execution to support various types of tags which cannot be allocated in the TRF. Although such structure of memory tags might be a good way to support diverse tag types using the existing memory architecture, it should be too slow if tags are frequently accessed from the tag space in the main memory. Therefore, to reduce the access latency, our PAU has an internal SRAM, called *tag cache* [27,44], for caching frequently referenced tags from the memory. In consequence, we would like to emphasize that our design for tag management with TRF and tag cache intends to empower PAU supporting fast tag lookups.

Since our system is implemented as a SoC, we have integrated our PAU to the multi-processor SoC platform, strictly following a platform-based design methodology. There are two design criteria that we have endeavored to satisfy when following the methodology for the development of our SoC hardware. First, we have tried to reuse as many existing modules as possible. They include various commodity IP cores, DDR memory and shared interconnects through which every module in the system is attached. Second, we have forced newly added hardware modules to comply with all the specifications required by our target SoC platform. For instance, ARM regulates that any IP module added to their platform obey the AMBA protocol. Therefore, in our implementation based on the AMBA platform, the interface to our PAU conforms completely to the AMBA protocol so that it can be connected to the host processor via the AMBA bus. In this sense, our solution differs from previous core-level ap-

proaches where their acceleration modules are added and connected to processors via custom lines or interconnects [19, 23, 44, 59, 87]. Also, all special logics customized for DPA are fully integrated into our PAU. This confirms our assertion that most hardware modules except the newly added PAU have been reused for our SoC implementation.

2.3.2 Execution Trace Communication

Now, we will discuss how the execution traces are transferred from the host to PAU through the system bus. To buffer the difference between the times for handling the assigned tasks on the two processors, we have implemented a dedicated queue, called the *trace buffer*, in PAU. In fact, other solutions based on separate processing units usually also need queues [19, 27, 44, 59] for similar purposes. However, others are rather core-level approaches thus demanding a change to the structure of their host CPU core or internal caches to some degree. On the contrary, by placing the buffer outside the host CPU and connecting it via a system bus, we preserve the original processor core architecture intact.

Figure 2.3 presents an example of the instrumented host code for a DPA method which analyzes the accessed memory addresses. It also displays the overall flow of trace transactions via the trace buffer between the host code and TPC in PAU. In the example, note that two additional instructions, being marked with boldface, have been inserted to the original code after instrumentation. They are added to generate two execution traces for memory addresses used by load/store instructions (traces #1 and #2). Suppose that the code is running on the host and reaches the code segment. Since *ld* instruction (1) is executed, the corresponding memory address stored in register %i0 should be gathered for DPA. As can be seen in the example, register %g4 is memory-mapped to the physical address of the trace buffer so that it provides a direct

way to store the trace in %i0 using *st* instruction (2). In a similar manner, a trace for memory address used by instruction (6) is also pushed into the trace buffer with the instruction (7). Finally, the stored traces are consumed by TPC for actual analysis task.

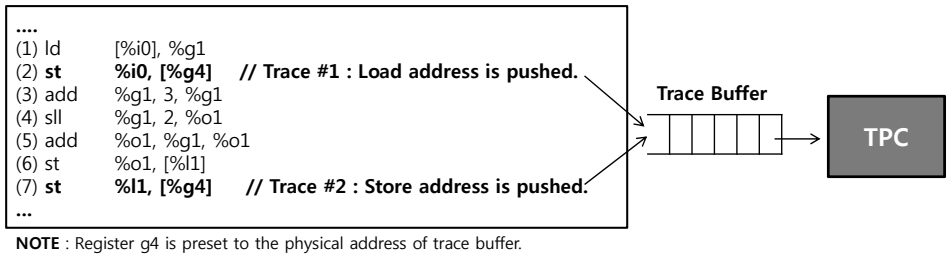


Figure 2.3 Execution trace communication

2.3.3 Synchronization and Multi-threading Support

In general, the approaches with separate hardware engine for DPA including ours should be able to handle the synchronization between the host CPU and the hardware engine. In our approach, the trace buffer is used to minimize the overhead of synchronizing the data transactions among these processing units by buffering the traces generated from the host, which helps the host continue its execution without being halted for the synchronization with PAU. However, such a basic synchronization mechanism based on the trace buffer may create a potential loophole in security for some DPA techniques that are to detect malicious attacks. For instance in DIFT, even if the host has just generated an important trace that is linked to a malicious activity, PAU may not recognize the activity until the trace is extracted from the buffer for the analysis in PAU. If the buffer is already filled with many preceding traces, the adversary may succeed in the attack long before PAU reaches the trace. To eradicate this

loophole, it would be necessarily required that the host CPU and PAU should be synchronized at every instruction [92]. However, as discussed in [36, 44, 75], such fine-grained synchronization may cause tremendous performance degradation in most cases. Thus in our implementation, the two computing units are synchronized at a more coarser granularity (i.e., at every system call), following the strategies of previous approaches [44]. The rationale for this decision is due to the fact that many compromised applications usually exploit system calls. For example, when an attacker wants to leak some sensitive data outside, the system call to open the network should be invoked. In this case, the data leak can be prohibited by checking the tag of data before sending the information through the network when DIFT keeps track of data flow during the runtime. Thus, we also utilize the system calls as an optimal granularity for synchronization in our architecture, to detect most malicious behaviors [44], and yet to substantially lower the performance overhead.

In our prototype implementation, every time a system call is invoked on the host, the OS kernel informs PAU of the event by sending a configuration command to PAU, and stops the execution of the monitored program. For synchronization on each system call, the host sets the *sync_syscall* register in the main controller. Once it is set, PAU consumes all traces left in the trace buffer and then reports its status to the host by sending an interrupt signal. Then, the host resumes its task after clearing the sync_syscall register.

Another important synchronization point we should consider is the context switch between applications. On the host CPU, many applications with different contexts are concurrently loaded. Thus, PAU should be notified of which process or thread is currently executed on the host CPU. In our work, these critical events are also announced to PAU by the OS kernel. Every time the OS scheduler is activated and a context switch occurs, the host notifies PAU this

event by writing the current process ID and the thread ID to the *current_PID* and the *current_TID* registers in the main controller, respectively. By doing so, PAU can identify the current process and thread ID on the host.

2.4 Tag Processing Core

In this section, we will explain the detailed design of TPU, the key component of our PAU, which is a processor core that enables us to write the software code for the DPA task in our approach. We will first describe the ISA of TPC whose mission is to support a wide range of tag-based DPA techniques. Then, the microarchitecture of TPC will be discussed.

2.4.1 TPC Instruction-Set Architecture

Basically, the TPC ISA is extended from a simple RISC ISA so that the general structure of software can be constructed with the ISA. Then, several types of instructions are added to the ISA, which perform the specialized analysis operations that are commonly inherent in the tag-based DPAs listed earlier. We will explain the data types handled by TPC and the types of instructions.

Data Types

In the TPC ISA, three types of data are supported to construct analysis task software; (1) tag, (2) general and (3) execution trace. Many DPA techniques typically associate a tag (that is meta-data) with each piece of state in the monitored program [27]. Thus, it is very natural to support the tag data type in our TPC design. Many details for the tag type were, in fact, discussed in Section 2. With the TRF and tag space in memory, TPC carries out a variety of tag operations to update and check the tags.

The general data type is necessary to construct a program structure for supporting tag operations. For example, to organize a loop structure that iterates the execution of a code segment for processing tags in an analysis code, there needs a set of data to control loop iterations such as loop indices and temporary variables. To express this type of supportive operations (e.g., loop iteration control) in the DPA algorithms, we provide the general data type for our TCU ISA. In the analysis code, the operands of the general type are distinct from those of the other two types in a sense that they do not contain any analysis specific information like tags or execution traces. Therefore, they are stored in a separate register file, called the *general register file* (GRF). During code execution, they must be loaded from memory to the GRF before being processed.

Lastly, the execution trace type is for the traces delivered from the host program. As stated earlier, the execution traces which contain the runtime information of the host are delivered to the analysis hardware engines such as our PAU. During the execution, TPC in PAU consumes the traces in order to follow the program execution flow and receive runtime traces which are not determined at instrumentation time. We assign the traces a different data type in order to distinguish the operations on them in the code from those on the other types of data (i.e., general and tag). For this reason, a trace in the trace buffer in PAU is regarded as a data of the execution trace type, which is accessed by a specific set of instructions. For example, in DIFT, the results of branches and the memory addresses accessed by load/store instructions are delivered to the trace buffer as the execution traces. In TPC, the traces are regarded as the data of execution trace type and processed by the special instructions to recognize the behaviors of the host. The further details will be given in the next subsection.

Instruction Types

To support the analysis tasks for DPA, we have designed four types of instructions in our TPC ISA; (1) general, (2) tag ALU, (3) tag load/store and (4) trace handling. The first set of instructions corresponds to those in a RISC-style instruction set to organize general program structure. In fact, it is directly matched to the general data type and gives our PAU the general programmability to construct analysis task software. The general group includes general ALU operations, load/store, data movement and branches, as shown in Table 2.2. They usually make use of GRF as the operands for computation and access memory space to load/store data. In addition to them, there are several instructions newly added for data movement between GRF and TRF. For example, *mov.tg* instruction moves the data in TRF to GRF. Then, the tag value can be manipulated by general instructions for the purpose of analysis and written back to TRF with *mov.gt* instruction. These move instructions widen the way to deal with the tags so that the degree of programmability in TPC ISA can also be improved.

Instruction Type	Sub-Type	Instructions	Example	Action
General	ALU/Data Movement	add,sub,mov, …	add R2,R3,R1	R1 = R2 + R3
	Load/Store	load	load [R2],#4,R1	R1 = Mem[R2+4]
		store	store R2,[R1],#4	Mem[R1+4] = R2
	Branch	beq,bneq,jump, …	beq #imm	PC = PC+#imm
	Data Movement to TRF	mov.gt	mov.gt R1,T1	T1 = R1
	Data Movement to GRF	mov.tg	mov.tg T2, R2	R2 = T2
Tag ALU	Register-Register	add.t,sub.t,and.t,xor.t, …	xor T2,T3,T1	T1 = T2 xor T3
	Register-Immediate	addi.t,subi.t,andi.t,xori.t, ...	addi.t T1,#1,T1	T1 = T1 + 1
	Tag Check	cmp.g	cmp.g T1,R1	compare T1 with R1
		cmp.t	cmp.t T1,T2	compare T1 with T2
Tag Load/Store	GRF Load/Store	load.g, store.g	load.g [R2],R1	Mem[TLB(R2)] = R1
	TRF Load/Store	load.t, store.t	load.t [T2],T1	Mem[TLB(T2)] = T1
Trace Handling	Trace Movement	mov.bg	mov.bg trace,R1	R1 = trace
		mov.bt	mov.bt trace,T1	T1 = trace
	Trace Compound ALU	add.tc, sub.tc, or.tc, …	add.tc trace,T2, T1	T1 = T2 + trace
	Trace Compound ALU/Load	add.tcl, sub.tcl, or.tcl, ..	or.tcl T2, [trace],T1	T1 = T2 | Mem[trace]
	Trace Compound ALU/Store	add.tcs, sub.tcs, or.tcs, …	or.tcs T1,T2,[trace]	Mem[trace] = T1 | T2

Table 2.2 Overview of TPC instruction-set

Many DPA schemes need to operate on the tags associated with processor registers. Thus, for these types of operations, TPC ISA includes tag ALU instructions that perform the operations among the register tags, as shown in Table 2.2. These instructions are functionally similar to the ALU instructions for the ordinary data except that their operands come from TRF. This type of instructions might be most frequently used in analysis software because tag updating and checking are the kernel parts of most DPAs.

On the other hand, in case of the tags located in tag space, they should be loaded to the registers for computation. To access the tag space, two types of tag load/store instructions are supported in TPC ISA. As given in Table 2.2, the GRF and TRF load/store instructions take operands from GRF and TRF, respectively. In most cases, analysis software performs TRF loads/stores to propagate tags between registers and memory locations. However, as in reference counting, DPA should update their tags located in the tag space without interacting with TRF. In these cases, a GRF load or store is useful because it does not pollute the status of TRF which contains the meta-data for processor registers. It is noteworthy that the tag load/store instructions are different from the ordinary load/store ones in that they use the *tag TLB*. In order to efficiently manage tags in memory, PAU employs the tag TLB proposed in Harmoni [27] that translates a data address to a tag address. By utilizing the specialized logic, the tag load/store instructions can be performed with the low address translation overhead.

Lastly, the execution traces from the host should be handled in PAU to follow the program execution flow. For this purpose, the trace handling instructions are provided. Recall that the traces are located in the trace buffer and accessed sequentially in order. In the trace handling instructions, the buffer is regarded as a register. To move a trace from the buffer to GRF/TRF, PAU sup-

ports two types of move instructions; *mov.bg* and *mov.bt*. The move instructions can be used when the trace should be further manipulated or interpreted to extract the required information. Also, there are a set of instructions, called the *compound instructions*, which take a trace as an operand. They substantially reduce the number of instructions to be executed when a trace from the buffer is used as an operand in tag updating. For example, in DIFT, memory addresses of store instructions are transferred to PAU as execution traces. In this case, the trace would be used as a destination operand in tag computations. If we would not have the compound instructions, the computations should require five TPC instructions as follows.

- instruction executed by the host : st [%g1], %g2
- execution trace : address value in register %g1
- tag propagation rule : Tag[Mem_addr[%g1]] = Tag[%g1] | Tag[%g2]
- DIFT analysis code in PAU :

    1. mov.bg R7 : trace movement to GRF (address in %g1)
    2. mov.tg T1, R1 : tag movement from the TRF to GRF
    3. mov.tg T2, R2 : tag movement from the TRF to GRF
    4. or R3, R1, R2 : tag propagation to a temporary register
    5. store.g [R7], R3 : update memory tags

A compound instruction, *or.tcs*, can substitute for the set of instructions. When it is executed, the address in the trace is used as a store address for tag space, and the tags in the two registers (T1 and T2) are propagated to the memory tag, while it increases the analysis performance. Because many analysis

schemes tend to directly associate the traces to their tags during execution, the instruction set is a good way to support them.

2.4.2 TPC Microarchitecture

In this subsection, we will show the microarchitecture of TPC for the ISA design described in the previous subsection. Figure 2.4 represents the internal block diagram of TPC. In order for TPC to launch tag computation, two preliminary conditions must be met. First, the analysis code associated with the instrumented host code has been created and located in the main memory. Second, the host CPU has begun the host code execution and deposited execution traces into the trace buffer, as demonstrated in Figure 2.3. As soon as a trace arrives, a notification signal is sent to TPC. Upon receiving the signal, TPC reads trace entries one-by-one from the trace buffer.

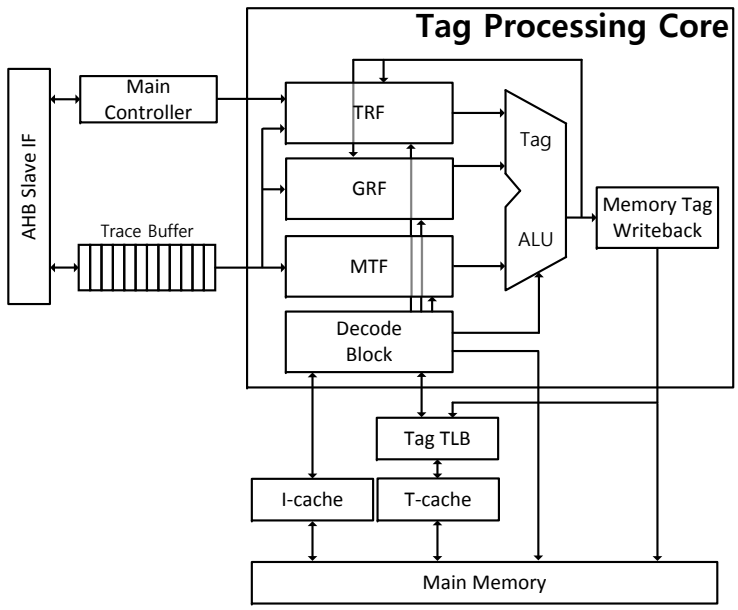


Figure 2.4 TPC microarchitecture

In order to function as a programmable processor, TPC has many general components of RISC architecture with a three-stage pipeline: (1) fetch-decode, (2) execution, and (3) write-back. At the first stage, TPC code is loaded from the memory and decoded by the *decode block*. Since main memory is normally implemented with external DRAM devices, off-chip memory access latency can be a serious performance bottleneck. To alleviate this problem, we have implemented the *instruction cache* between TPC and main memory.

At the next stage, TPC fetches operands from the two register files, and accesses the tag space with the tag TLB and the tag cache. At the same time, TPC schedules the *memory tag fetcher* (MTF) unit to fetch the tags in memory according to the memory addresses in the trace buffer, in order to support the trace compound instructions. An operand of the general type is also loaded from the main memory at this stage. In our current prototype, there is no dedicated cache for the data of general type mainly because the operations on general data are relatively few as being compared to the other types of operations. However, if a developer wants to cache a set of general data, it is also possible to map them to the tag space so that they can be cached in the tag cache. After all operands are ready, they are then forwarded to the tag ALU that takes these tags as the operands to conduct tag computations or other general ones.

At the last stage, TPC updates the result back to either the register files or the memory space, depending on the executed TPC instruction. Just in case the tag cache is updated with new data, we have implemented a write-through cache scheme in order to keep the consistency of data stored both in this tag cache and the tag space inside main memory. Once TPC has completed the execution of all instructions fetched and there is no trace from the host, it will be idle waiting for new traces filled into the buffer by the host. If not, it reiterates the normal execution procedure as described so far.

2.5 Case Studies

The programmability of the TPC ISA offers developers a good capability of implementing a variety of their DPA schemes with flexibility in software on our PAU. In this section, as case studies, we will discuss how several well-known DPA techniques can be realized on our prototype as software codes that are composed of the TPC instructions. As examples, we picked three techniques; DIFT [62], uninitialized memory checking (UMC) [93] and bound checking (BC) [22]. After briefly introducing the idea of each DPA scheme, we will discuss our DPA implementation on PAU.

2.5.1 Case Study 1 : DIFT for Data Leak Prevention

Background of DIFT

To protect the confidential data inside computing devices, an approach called data leak prevention (DLP) has been proposed. In the approach, security policies defining critical information and the corresponding actions (that is, deny/permit) on the specific output channels are forced to prevent any critical information from flowing into the outside of devices. A common way to realize DLP has been to use DIFT [31, 99], one of the widely used DPA techniques. This analysis scheme sets up rules to tag (or taint) internal data of interest and keeps track of the taintness of their tags throughout the system [44]. At run time, every data derived from the one with tainted tag has its tag tainted. An alarm will be triggered as soon as any of the tainted data involves in potentially illegal activities, such as pointing inside the code or being included in a data stream on the output channels [31].

When DIFT is employed for DLP, the first step is to tag or taint as sensitive the input data from sensitive sources like confidential files. Then, through code

execution, the data tags are then also propagated by tagging as sensitive the data derived from those with tainted tags, following the tag propagation rule of DIFT. If the code makes an unauthorized attempt to leak any of tainted data [73], a security exception will be raised to announce the existence of data leak.

Tag Initialization and Check Procedures

As introduced, DIFT uses the tags to indicate the taintness of the data. To support the tag-based analysis in this case study, we assign a 1-bit tag for every host processor register and store it into our TRF in TPC. Each 1-bit assigned to a register in TRF represents whether or not the corresponding processor register currently holds sensitive data. Likewise, one bit is assigned for each word in memory, and these bits are all arranged in the tag space, in a similar way to that suggested in [92].

In general, we can divide the tasks of DIFT for DLP into the three stages: tag initialization, propagation and check. In this study, the tag initialization and check stages are executed by the OS kernel in the host processor. Depending on whether data originates from a sensitive source, the kernel initializes its tag by setting the bit on or off. Just before data is transferred to an output channel, the kernel checks its tag to decide if the data transfer is safe. We will discuss the tag propagation stage later in order to first focus our discussion on these two stages which require a close interaction between the kernel and PAU.

On a computing device, sensitive sources include GPS, files with confidential contents and SIM cards with private information. In specific, in this study, we focus on the confidential files on the system as our sensitive sources and the kernel maintain the list of them. To monitor every access of an application to any file in the system, we modified `open` system calls in our Linux prototype

system. When one of applications opens a file by invoking the system call, the kernel determines if the file is in the list. If so, the file pointer will be tainted by setting its bit on. For this tag initialization, the kernel function *tag_init* is invoked. In our system on the host processor, the function is implemented as a device driver interacting with our PAU. Its task is reporting to PAU the location (i.e., register number or memory address) of the data that must be tainted. Depending on its type, the location is written to either *source_taint_reg* or *source_taint_addr*, both of which can be configured via changing the values of memory-mapped configuration registers in the main controller of PAU. Then PAU responds the report from the kernel by tainting the tag for the location.

For the tag check stage, we also have embedded a new function *tag_checking* into the system calls involved in network packet generation. When data is about to be transferred outside as a network packet through an output channel, this kernel function checks the data tag with the assistance of PAU. As the first step of this check, the function writes the data location into either *sink_taint_reg* or *sink_taint_address* in the configuration registers, similarly to the tag initialization stage. Then it sends to PAU the inquiry of the current tag value at this location. Upon receiving the inquiry, PAU retrieves the value from either TRF or tag cache, and interrupts the host to notify the result back to the kernel. Now the kernel knows whether or not the data of interest is from sensitive sources.

Tag Propagation

As explained in Section 2, in our approach, the time-consuming part of DPA is delegated to PAU to relieve the burden of the host processor and it corresponds to the tag propagation computations in DIFT for DLP. To carry out the propagation task on our PAU, TPC code includes propagations rules and operands extracted from the original program run on the host processor. When

our in-house instrument tool generates the code, most required information like register operands can be statically extracted and embedded in the generated code. But some dynamic information that can only be resolved during code execution is still missing in the generated code. In our DIFT implementation, such information includes (1) an execution path of the original program and (2) memory addresses of load/store instruction. Currently, this missing information is supplemented and sent as execution traces at run time by the host processor to PAU, hence helping PAU have the enough information to track tag propagation at any circumstance.

Figure 2.5 shows a segment of the original program in (a), its associated pseudo propagation code in (b) and the realized TPC code in (c). As can be seen from (b), a pseudo instruction is comprised of a propagation rule and operands. They are to specify the semantics of tag propagation by the matching instruction in (a). For instance, the third DIFT instruction in (b), which is parallel to the *sll* instruction in (a), has "%o1 and %g1" as operands and "copy from the right tag to the left" as a rule. When the original instruction is executed, so does the pseudo instruction and thus if the tag of %g1 is tainted, %o1's tag will also be tainted because of the 'copy' tag propagation rule.

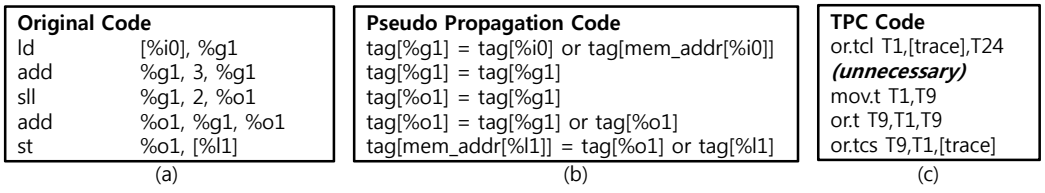


Figure 2.5 A TPC code example for DIFT computation

In the analysis task of PAU, the propagation rules are expressed by the TPC instructions as given in Figure 2.5 (c). For two load/store instructions in the host code, the propagations are processed by the compound instructions

since they make use of the trace in the trace buffer to figure out the location of tags. As seen in this example, our compound instruction can reduce the number of instructions required for the trace handling. For the other ALU operations in the host, the tag ALU instructions are used to propagate the tags between the tag registers. However, for the second instruction *add*, the corresponding TPC instruction is omitted because it does not change the tag status of PAU. In our software implementation, we have made efforts to remove these unnecessary propagation operations, being empowered by the programmability of TPC. The TPC codes are generated by our in-house instrument tool and allocated to TPC code memory region before the execution.

During the host program execution, PAU expects execution traces from the host processor to compensate for the missing dynamic information that is indispensable for correct operation. The load/store addresses can be easily gathered into a trace since they are readily computable from the host program at run time. As for the execution path, we may express it with a set of basic blocks and edges that connect them. Thus in our implementation, we assign every basic block a *unique identification (ID)* number, and during code execution, let the host processor deliver the ID of a block to PAU so as to pinpoint the exact block that the host execution path currently comes to.

As seen in Figure 2.1, the original program installed on the host has to be instrumented to enable communication with PAU for orchestrating analysis operations in our solution. Figure 2.6 presents an example of the instrumented host code generated from the original one in Figure 2.5. It also displays the overall flow of trace transactions for DIFT via the trace buffer between the host code and TPC. In the example, note that four additional instructions, being marked with boldface, have been inserted to the original code after instrumentation. They are added to generate three traces, one for the current

basic block information (trace #0), and two for memory addresses used by load/store instructions in the same block (traces #1 and #2). Suppose that the code is running on the host and the execution path comes to this block **LL5**. Then, *mov* instruction (1) is first executed to initialize register %g3 with the basic block ID. As can be seen in the example, register %g4 is memory-mapped to the physical address of the trace buffer, thereby providing a direct way to store the trace in %g3 using *st* instruction (2). In a similar manner, traces for memory addresses in the basic block are also pushed into the trace buffer with the instructions (4) and (9). The stored traces are consumed by TPC for tag propagation.

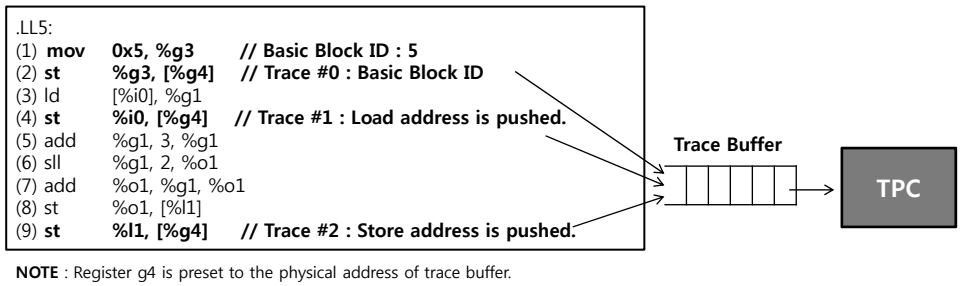


Figure 2.6 Execution trace communication for DIFT

In order to conduct the analysis task for DIFT with the basic block ID, we decomposed the TPC code into multiple regions, each containing a single basic block of TPC instructions, as Figure 2.7 depicts. Each region includes a header for its basic block. The basic block header holds the useful information for PAU (e.g., the number of TPC instructions and the number of load/store). At the beginning of the TPC code region, there is a lookup table, called the *basic block jump table*, which is used to access every basic block in the TPC code. During execution, when TPC receives a trace indicating a basic block ID, it accesses the basic block jump table. Then, it jumps to the address and finds the TPC

instructions within this basic block. After finishing the block, TPC will find another trace in the buffer and continues the analysis if the buffer has one.

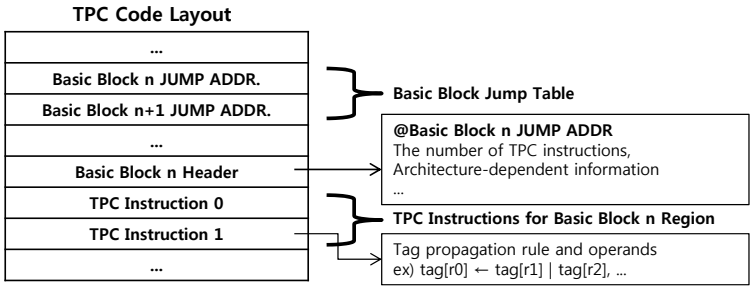


Figure 2.7 TPC code layout

2.5.2 Case Study 2 : Uninitialized Memory Checking

Background

As the second implementation example of our case study, we chose UMC which was firstly proposed in [93]. The objective of this DPA technique is to detect a read access to the memory region where initialization is not performed yet. Since the read event to an uninitialized location causes a memory error which is often exploited by attackers as a security hole, it is vital to detect and remove such cases in program execution for security purposes [93].

In Figure 2.8, we depict the state transition diagram of the UMC scheme to explain the principle of the DPA. As explained in Table 2.1, UMC augments an 1-bit tag for every word in memory to indicate whether the corresponding location is initialized or not. When the system is reset, every memory location is assumed to be uninitialized. If a value is stored to a certain memory location, the state of the location is transited from *Uninitialized* to *Initialized* as shown in Figure 2.8. Once a memory location enters the *Initialized* state, both load

and store operations from/to the location are permitted . However, any load access to a location with *Uninitialized* will be called an error. Now let us explain how this memory checking model can be mapped to the tag-based DPA using PAU.

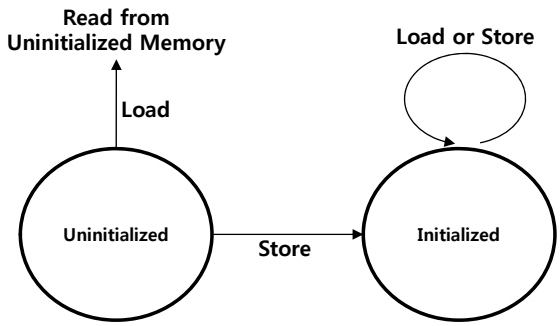


Figure 2.8 State transition diagram for UMC

UMC Implementation

To carry out UMC on PAU, we assign a 1-bit tag for every word in the application memory region and store the set of tags into the tag space in the main memory. As explained above, the tasks of UMC are mainly divided into two parts; tag initialization and check. In our UMC implementation, both the operations are composed of TPC instructions with the execution traces delivered from the host as input. For every memory write on the host, the write address is transferred to TPC from the host. Then, the tag corresponding to the address is set to '1' in order to indicate that the location is initialized. This is the tag initialization process. Likewise, when the host reads a memory location, the address is also delivered to TPC. At this moment, TPC checks the value of the corresponding tag and if it is not '1' (i.e., *Uninitialized*), an exception is raised to inform the host of an unallowable memory access.

Figure 2.9 illustrates a segment of the original example program in (a), its associated pseudo UMC code in (b) and the implemented TPC code in (c). When the original code is executed on the host, TPC code also runs in parallel to check whether the memory access rule enforced by UMC is violated or not.

For each load instruction in the host code, the tag check is performed by three TPC instructions. At first, a compound instruction, "mov.tcl" is used to read a trace in the trace buffer which contains the accessed address and load the memory tag corresponding to it. Then, a comparison between the memory tag and the register R1 is performed by the "cmp.g" instruction. To mark the *Initialized* state, the register R1 is set to '1'. Thus, if the comparison between the tag and R1 produces the "not equal (ne)" condition, it implies that the load instruction attempts to read an uninitialized memory location. For these cases, according to the rule of UMC, TPC jumps to an exception routine to trigger an alarm by sending an interrupt to the host. In this example, the label for the routine is named as "trigger_alarm". On the other hand, for each store instruction in the host code, the tag initialization is performed by the mov.tcs instruction. In our example, the register T2 is also preset to '1' to indicate the *Initialized* state. By writing the value (i.e., 1) to the memory tag of the delivered address, TPC carries out the tag initialization process.

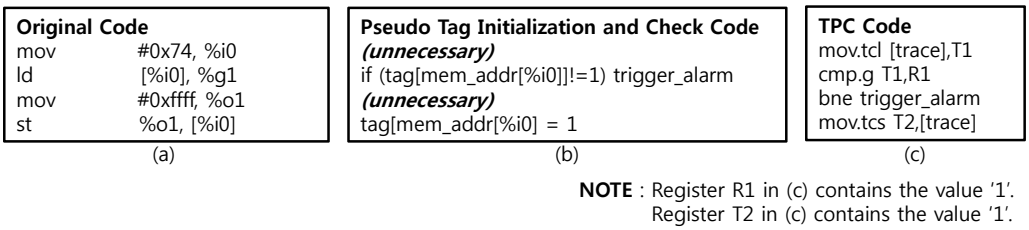


Figure 2.9 A TPC code example for UMC computation

2.5.3 Case Study 3 : Bound Checking

Background

As the last implementation example, we chose BC, which is a DPA technique proposed in [22]. The objective of this scheme is to check whether or not each memory operation with a pointer accesses the location within the legitimate range allocated by the program for the pointer. If it ever makes an out-of-bound access, BC reports the access as a memory error since it can be exploited as a security vulnerability.

The tasks of BC can be divided into three stages; tag initialization, propagation and check. To perform the procedure, BC augments the tags for both pointers and corresponding memory locations [27]. Whenever a memory region is allocated at runtime, BC initializes the tags for both the memory locations and the pointer to the starting address. In a high-level language like C/C++, special functions are provided for memory allocation, such as *malloc*. They usually take the size of the requested memory as input and return the starting address of the allocated region. Every time the functions perform their task, BC identifies the range of the allocated memory in the form of "[p, p+*size*)", where p is the starting address returned and *size* is the size of the allocated region [22]. Then, BC assigns the same tag value to both the tags of the allocated memory locations and the tag of the register which contains the returned pointer.

During the execution, the pointer tag is propagated to the other storage location in accordance with the propagation rule of BC [22]. On each memory instruction such as load or store, the register tag for the pointer is compared with the tag of the accessed memory region. Obviously, the two tags must be identical for in-bound accesses but different for out-of-bound accesses [22].

Therefore, only when the tags are identical, the memory access will be granted. Otherwise, BC reports the memory error.

BC Implementation

To implement BC in this study, we assign 4-bit tags for memory locations and pointers as in [27]. In Figure 2.10, we depict the host code in assembly level in (a), the pseudo tag initialization/check code in (b), and the pseudo tag propagation code in (c). In (a), a memory-allocation function, `malloc`, is invoked at line 6. The parameter of the function is the size of the requested memory and set at line 5 (in register %o0). After the `malloc` allocates the memory region, the starting address for the region is written to the register %o0 according to the calling convention. At this time, the host transfers two traces to the TPC; the size of the region and the value of the pointer. Then, TPC performs the tag initialization which assigns the same tag values to the tags for the allocated memory region and the tag for the corresponding register %o0 as shown in (b).

After that, during execution, the pointer tag is propagated to other registers or memory locations as shown in (c). The tag propagation rule of BC is almost the same to that of DIFT. Then, when the host attempts to access the allocated memory at line 17, with the delivered trace which contains the accessed address, TPC checks whether or not the tag of the pointer (the tag for %g1) is matched to the tag of the accessed memory location. If the both tags do not match, an exception is raised and the interrupt to the host is triggered.

Finally, when the host executes the deallocation-function `free` at line 27, the tags associated with the deallocated memory area are cleared. As discussed in [22], the pointers that were tainted for the deallocated region might not be cleared. This is because, with the uncleared tag, BC can detect the memory

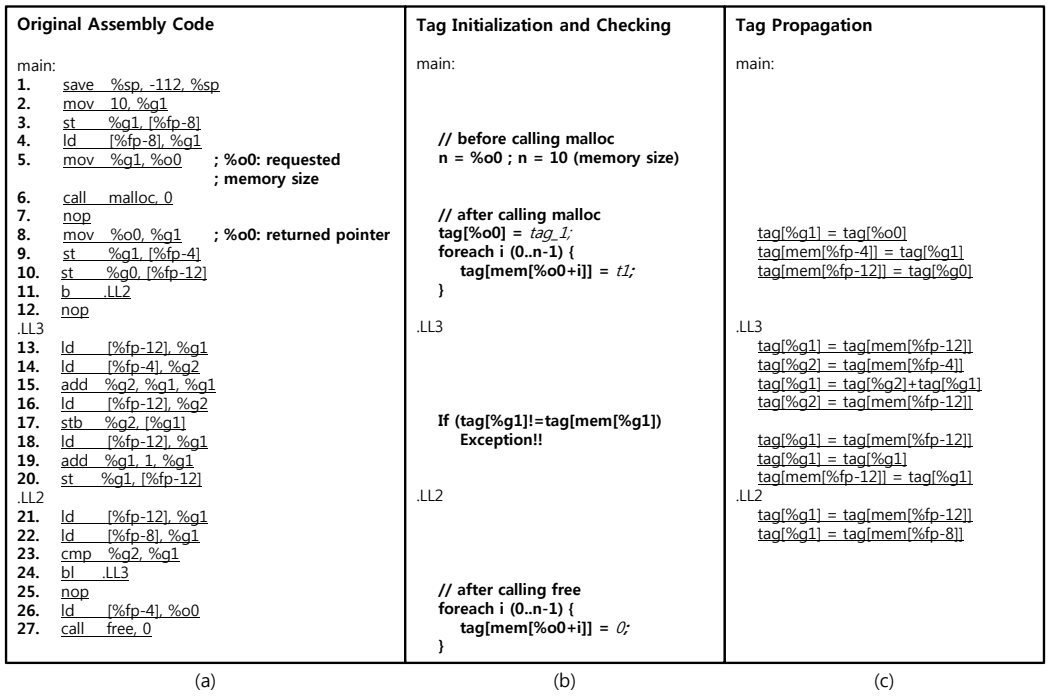


Figure 2.10 A pseudo code example for BC

accesses to the deallocated region by checking if the both tags have the same value. For our BC implementation, we do not describe the detailed TPC instructions that correspond to the pseudo code in Figure 2.10 because the most parts of the implementation are the same to our other DPA examples. For the tag initialization and the tag check procedures, our BC implementation is almost the same to the UMC implementation. On the other hand, for the tag propagation process, the most TPC codes used for DIFT were re-used for BC.

2.6 Implementing Optimizations for DIFT with TPC

In our case studies, we have clarified how different DPA techniques can be realized on PAU simply by programming the algorithms. In this section, we

will present another practical example where the programmability of TPC can be well exploited. Discussing the DIFT implementation in Section 5, we explained the basic instruction-level tag propagation for DIFT. However, since the instruction-level tracking incurs too much overhead, several previous studies on DIFT have centered their efforts on the overhead reduction by adaptively choosing coarser granularities (i.e., basic blocks or functions) [73, 104]. In this section, we will show that these optimizations can be adopted into our DIFT implementation with the programmability of TPC. By doing so, once an application is chosen to be monitored, DIFT algorithm running on TPC can find optimal code granularities of tracking operations for each different part of the application. In the followings, we will discuss how code analysis information is applied to help our DIFT implementation to adaptively choose optimal granularities for tag propagation within individual basic blocks or functions such that data leaks can be prevented while computation overheads are minimized. We will also describe how our PAU supports multi-level tag propagation efficiently in hardware.

In an attempt to choose optimal granularities for tag propagation within an application, we first divide application code into functions of three categories as follows, referring to the prior researches on DIFT [73, 104]:

1. Their output tags are independent of input tags.
2. Their tag propagation behaviors are known a priori and so summarized in a well-defined form.
3. None of the above.

For categories 1 and 2, tag propagation can be optimized by either skipping the computation completely or doing efficient *function-level* computation with

only a few TPC instructions and execution traces, thus relieving the computation loads from PAU. For the last category, exhaustive finer-grained computations are inevitable since intensive monitoring is mandatory due to the nature of these functions. Fortunately in our DIFT implementation, we can still hinge the optimization of these heavy computations on our PAU which helps us not only to accelerate *instruction-level* computation but also to enjoy faster *block-level* computation for some parts of the functions in this category. In the followings, we will discuss our optimization strategies according to these categories.

2.6.1 Function Level Tag Propagation Optimization

Given a function of category 1 or 2, the whole tag propagation can be virtually turned off even though a small number of TPC instructions along with traces still need to be executed to fulfill complete tag propagation for those in category 2. Since huge performance gain can be obtained via function-level tag propagation, we try to maximize it by classifying as many functions as possible into categories 1 and 2 during our offline binary translation. This classification can be done by adopting traditional static analysis [80, 104]. Another way to achieve it might be collecting a list of highly utilized functions encountered in applications such as library functions whose semantics are also well known and defined. For this purpose, we profiled a set of real programs in order to choose such functions that consume most time in them. When a function is found to be of category 1 or 2, we construct a function summary which is composed of the function name, TPC instructions and the code for execution traces that will be added to the original binary for the function. These summaries are created into the *function summary table* (FST). During binary translation with the original application, every function name in the code is brought to see if any function summary in FST has the name. If so, our instrument tool uses the information

in the summary to produce the optimized TPC code as well as the host code that is instrumented to generate execution traces.

We present a code example in Figure 2.11 to explain in more detail how our adaptive multi-level DIFT is applied. Figure 2.11 (a) shows the original application code, where the invocation to the `malloc` function at line (3) takes the size as an input and returns a pointer to the allocated memory space as the output. A simple analysis on this function may easily reveal that the output tags cannot be derived from the input one because the resulting pointer and memory locations are not data dependent on the input size. As a consequence, the function should belong to category 1 by definition, and so its name has to be found in FST. As shown in the example, we see that our instrument tool produces neither code for traces nor for DIFT to save our computing resources, according to our optimization policy.

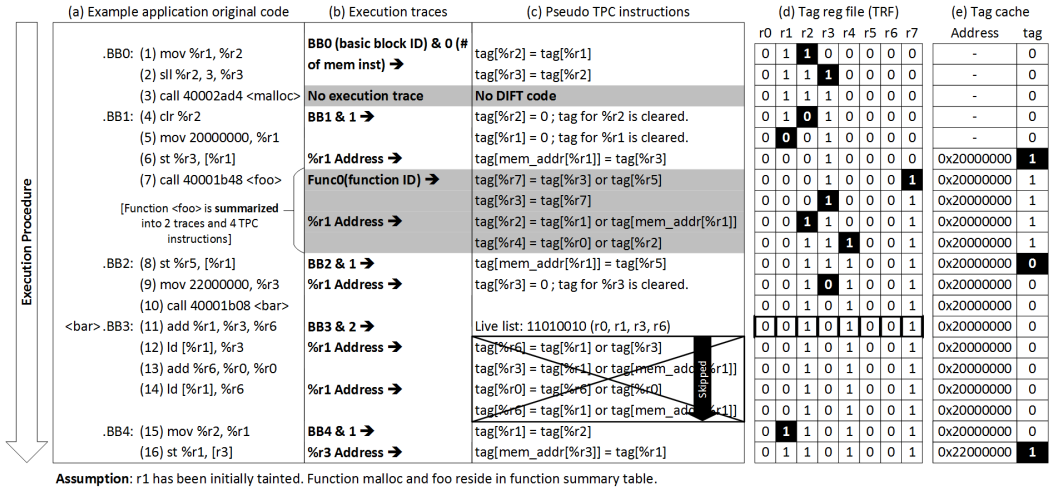


Figure 2.11 An example for adaptive multi-level tracking

We assume that the function `foo` at line (7) is of category 2. Then, we can apply function-level DIFT to the function, as we explained. Therefore, small

numbers of execution traces and TPC instructions are enough for complete tag propagation within `foo`. The figure shows that our instrument tool generates only two traces and four TPC instructions referring to FST. Figure 2.11 (b) and (c) respectively depict execution traces and TPC instructions generated after the multi-level tracking optimizations. Now notice that, in (b) for `foo`, the *function ID* is assigned in the first entry of the trace buffer. Similarly to basic block IDs in Section 5, a function ID is used to point PAU at the position where its TPC code starts to execute.

The shaded regions in Figure 2.11 represent the function-level tag propagation for the functions of categories 1 and 2. This clearly assures our argument that function-level optimizations improve the performance of both the host processor and PAU by drastically reducing or eliminating the computation loads due to execution traces and TPC instructions.

2.6.2 Block Level Tag Propagation Optimization

Although function-level tag propagation has a great affirmative impact on performance, all functions cannot take such benefits. Not surprisingly in real applications, a majority of functions fall into category 3. In principle, these functions necessitate instruction-level propagation, which will slow down the processing speed. To mitigate the overhead and further improve the DIFT performance, we exercise a coarser-grained tag propagation on some basic blocks dynamically during code execution. The optimization technique on block level was proposed in LIFT [73]. The basic idea is that the whole tag propagation in a basic block can be safely precluded if all the live-in/out rags of registers and memory locations into/from the block are untainted (i.e., the tag bits are all set off) at the boundary of the basic block. In LIFT, the decision is made just before the host CPU enters the entry of each block at run time.

In our work, we also implement and apply the same optimization scheme on our DIFT. The main difference between ours and LIFT is that the decision for every basic block is performed by PAU in our work. Consequently, this makes the host CPU to be liberated from the decision task, which otherwise slow down the host performance due to the computation overhead required for the task (e.g., instructions for managing and checking the relevant tags, context switches for preserving the host program's states). That is, whether or not a basic block satisfies the above conditions, the host proceeds with its normal execution of the instrumented binary for this block, as described in Section 5. At the same time, TPC would extract from the trace buffer the execution traces that were issued from the host at the beginning of the current block. Recall that whenever TPC enters a new basic block, it reads the block ID from the first trace to execute the TPC instructions in the block. At this moment, it will examine all live data tags crossing the block boundary. If none is tainted, TPC just skips the execution of this block and be ready to extract new execution traces for the next block as directed by the host.

To enable this block-level optimization, we need to collect a summary about live tags around each block. For this, we have augmented our instrument tool to support live range analysis that identifies the live register tags coming into/out of every basic block, and puts them into the live list attached to each block. Assuming that n_r is the total number of registers, the live list is a bitmask of the size n_r bits. If a bit is set to 1, this represents that the corresponding register tag is alive at the entry of the basic block. By simply reading this list, TPC can determine the liveness of all register tags with ease. Contrary to the case of register tags, we do not apply static analysis to identify the live tags of memory locations obviously because exact memory addresses referenced in the code cannot be statically known in most cases. Therefore in our system, TPC

collaborates with the host to dynamically figure out the liveness of memory tags at run time. To attain this objective, it accepts from the host all memory references in a basic block through the trace buffer.

Figure 2.12 displays a small decision logic that is vital to TPC's taint check of each live register or memory tag, which in turn collectively leads to the final decision on the applicability of block-level optimization to the current block. This logic determines whether live register tags are tainted or not by performing bitwise AND operation between the live list from the TPC code and the contents of TRF which is also an n_r-bit bitstream of tags of registers. If the result of this operation is zero, all the live register tags are untainted. Live memory tags should also be considered by accessing the tag cache with the addresses stored in the trace buffer. As shown in Figure 2.11, the number of memory addresses to be handled in the block is delivered as the execution trace (basic block ID and the number of memory addresses are encoded to a word). Thus, TPC can know how many memory tags should be considered for the decision. Since the tag cache has a single read port, it may take multiple cycles to load all memory tags depending on the number of memory addresses in the trace buffer. Finally, if all the live tags are declared untainted, TPC bypasses time-consuming instruction-by-instruction tag propagation inside the block, just waiting for the next direction from the host.

In Figure 2.11, we can see an example of block-level tag propagation within the function `bar`. Let us assume that `bar` is of category 3. Then, this function would not be found in FST as defined earlier. Instead, the instrument tool generates the TPC code in which the live list for each basic block in `bar` is attached to its entry. The tool also produces an instrumented code that will run on the host. Suppose that the host is about to execute the basic block **BB3** at line (11). At that time, TPC is ordered to execute the TPC code at

the same line. As the first step, TPC has to extract the live list from the code, which is a bitmask 11010010 in this example. Then, it will compare it with the contents of TRF shown in Figure 2.11 (d). Simultaneously, the memory tag at address 0x20000000 is also retrieved from the tag cache shown in Figure 2.11 (e). We can successfully verify that all the tags are untainted in this example. This means that the entire tag propagation in the basic block can be safely omitted. Figure 2.11 exhibits that all TPC instructions for the block **BB3** are crossed out to indicate that the entire code execution on TPC is bypassed. This example shows that our block-level optimization can enhance the DIFT performance with the cooperation of software analysis and hardware support.

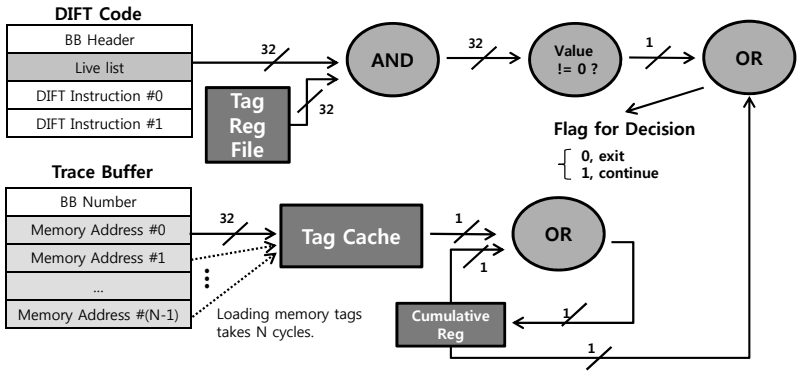


Figure 2.12 The decision logic for block-level optimization

2.7 Experiment

2.7.1 Prototype System

To evaluate our approach, we have developed a full-system FPGA prototype, where the host processor is the SPARC V8 processor, a 32-bit synthesizable core [55] which uses a single-issue, in-order, 7-stage pipeline. It has separate 16K-byte 2-way set associative instruction and data caches. The trace buffer has

been implemented to accommodate at most 64 execution traces (i.e., 64x32bit) passed from the host. The architecture of our PAU follows the description in Section 3 and 4. It has the tag cache which is a 512-byte, 2-way set-associative cache with 8-byte cache lines, and the instruction cache which is a 4K-byte, 2-way set-associative cache with 32-byte cache lines. The bus compliant with AMBA2 AHB protocol [51] is used to interconnect the all modules in our prototype system. Linux 2.6.21.1 is used as our OS kernel and a small portion of it has been modified to provide supports for our hardware engine as described before. Based on the parameters for the prototype as described above, we synthesized our DPA engine and verified it on a FPGA prototyping board with a Xilinx XC5VLX330 FPGA and 64MB external SDRAM.

2.7.2 Synthesis Results

When our hardware engine is employed in the systems that have severe resource constraints such as mobile devices, the area and power budgets of PAU are also strictly limited. Thus, in the systems, the area/power efficiency of the hardware engine is the foremost priority. In order to assess the area efficiency of our PAU, we quantified the resources necessary for PAU including the tag cache, instruction cache and trace buffer, in terms of gate counts using Synopsys Design Compiler [38] with a commercial 45 nm process library. In Table 2.3, the number of gates required for each component of PAU is described and compared to those of the baseline system including the host processor. The total area overhead for PAU is about 14.47%, as compared to the baseline system. Considering that our host processor is a very small SPARC RISC processor, we assure that the area overhead for PAU is not critical to be deployed in the commercial platforms.

As shown in the table, PAU can be divided into four parts; the compo-

nents for tag computation which might correspond to the DPA engines in the hardware only approach [27], the components added for the programmability, the trace buffer and the remaining parts such as AHB interface and interrupt generator. To support a wide range of DPA schemes with the programmability, PAU requires additional resources as described in the table (especially for the I-Cache). Although the amount of resources seems to be substantial, the total area overhead of PAU is not so huge as stated above.

To estimate the power consumption of PAU, we simulated our framework on Modelsim [37] and run the power estimation tools in Synopsys Design Compiler [38] using the simulation result as an input vector. As a result of experiment using a commercial 45 nm process library, the power consumption of PAU is estimated to be 224.2 mW at 1 GHz operating clock frequency. Since it is acceptably small when compared to the power consumption of SPARC host processor (940 mW at 1 GHz), our PAU can be deployed in the commercial SoC platforms which have the limited power budget.

2.7.3 Performance Evaluation

We have measured the performance improvement of our hardware engine over the previous approaches by choosing applications from the mibench benchmark suite [40] and comparing in performance with the four configurations. In this experiment, the configuration NC stands for native code which executes the original codes on the host CPU with DPA disabled. This is used as baseline, and all the other configurations are set with DPA enabled. For SWD, not only the code of original program but also the code for DPA are executed on the host core. Thus, whenever the DPA procedure is needed at a certain point of the program, the host invokes the function which performs the tag computations. For MPD, to improve the performance, the tag computations for DPA

Category	Component	Gate Counts
Baseline System	SPARC V8 Core (Host Processor)	1761079.777
	Bus components (AHB Buses + AHB/APB bridges)	2137.61
	Memory Controller	3812.52
	Peripherals (TIMER, UART, and etc.)	3304.15
	Total Baseline System	**1770334.057**
PAU	Tag Register File (TRF)	957.4992
	Decoder	2305.6902
	Tag Cache	13253.2245
	Tag ALU	4932.274
	Tag TLB	10266.833
	Total Resources for Tag Compuation	31715.5209
	General Register File (GRF)	950.1425
	Main Controller	1339.0524
	Memory Tag Fetcher (MTF)	13253.2245
	I-Cache	203811.3338
	Total Resources for Programmability	219353.7532
	Trace Buffer (16 x 32-bit)	3425.1589
	ETC (including AHB Slave Interface)	1683.0324
	Total Resources for PAU	**256177.4654**
	% PAU over Baseline System	**14.47%**

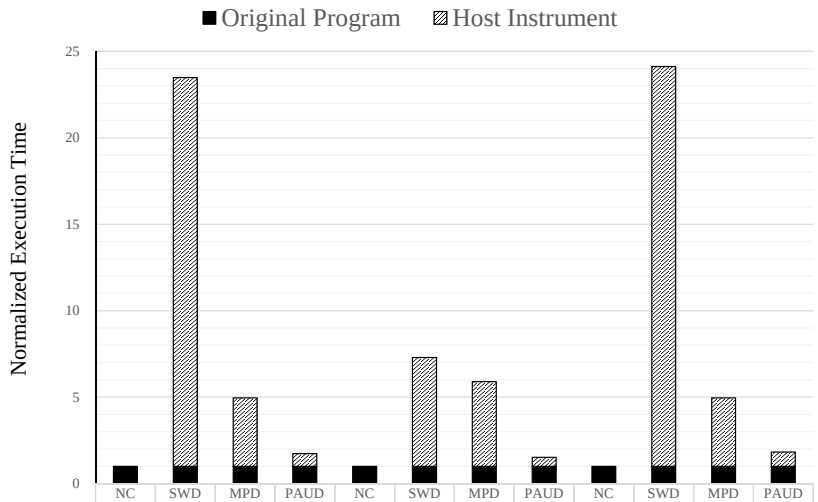
Table 2.3 Synthesis result

are offloaded to another general purpose core which is dedicated for the analysis, called *analysis core*. However, the handling codes for sending the execution traces still needs to be invoked on the host since we assume that the multiprocessor approach in our experiment does not have the dedicated hardware queue such as the trace buffer. The register file of the analysis core acts as the shadow register file to store the tags of the host registers, as proposed in [59]. For this reason, the analysis core should preserve and restore the states of the registers when the DPA needs to use the registers for other general computations, such as trace handling. Lastly, for the configuration PAUD, the tag computations are offloaded onto our PAU. With the help of the dual register file architecture (that is, GRF and TRF), PAU can remove the overhead of the context

switches which is paid in MPD, because it uses the registers of the GRF for general computations. Also, the host can reduce the overhead for transferring execution traces since the trace buffer can be accessed by simply storing the traces to the predefined memory address, instead of executing the codes for trace communication.

In Figure 2.13, we depict the performance comparison among the four configurations. We have measured the average of host execution time for eight applications in mibench (i.e., dijkstra, bitcnt, rijndael, sha, blowfish, strsearch, patricia and qsort) and they are normalized to that of NC. SWD runs on average 7.3-24.1 times slower than NC because the additionally instrumented codes for DPA are performed by the host. In MPD, an additional general purpose core is dedicated to perform DPA computations but the slowdown of the multiprocessor approach reaches up to 4.9-5.9 times of NC due to the inefficient structure of general cores. To the contrary, PAUD substantially cuts the overhead down to 52.0-82.8% of NC for the three DPAs through the acceleration with PAU. It is 4.7-13.6 times faster than SWD. Even from MPD, PAUD enhances the analysis performance up to 2.7-3.8 times. The results clearly shows that our approach can be effective in leveraging DPA performance.

So far we have assumed that PAU and the host processor operates at the same clock speed, but this assumption might not apply to recent systems in which the host operates far faster than other modules [5, 77]. In this environment, our host would swiftly produce execution traces at a rate more than PAU could consume. This may cause the host to stall frequently, thereby slowing down the overall program execution. To remedy this problem, PAU should either operate at a higher frequency or have a bigger trace buffer that may accumulate more traces. However, these remedies might be unacceptable since they raise hardware costs. Therefore, our PAU should be able to tolerate the



	NC	SWD	MPD	PAUD	NC	SWD	MPD	PAUD	NC	SWD	MPD	PAUD
	DIFT				UMC				BC			
Host Instrument	0.00	22.48	3.97	0.73	0.00	6.29	4.90	0.52	0.00	23.12	3.96	0.83
Original Program	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Figure 2.13 Comparison of execution time (normalized to native)

performance gap between the two processing cores.

To certify that our PAU can circumvent this very problem, we conducted an experiment with the configuration PAUD, under the condition that the host processor runs 2 to 8 times faster than PAU as done in [44]. Figure 2.14 depicts the execution times of PAUD normalized to NC when the performance gap is increased. As shown in the figure, the execution time of PAUD is affected by the three component; the execution time of original program, the overhead incurred by the instrumentation on the host code and the synchronization overhead due to the performance gap between the host and PAU. For the three DPA techniques, when PAU and the host operate at the same frequency (see 1X in Figure 2.14), the performance of PAUD is affected only by the host instrument. That is, PAU can keep up with the processing speed of the host at this condition. However, as the performance gap increases, the synchronization overhead is also

increased since the relative computation power of PAU decreases. Nevertheless, the amounts of increased overhead are less than 30% for the three DPAs even when the performance gap reaches up to eight times. The results imply that our PAU is applicable to a broad range of platforms even when the performance ratio between the host processor and PAU becomes increased, with the help of the specialized architecture of PAU.

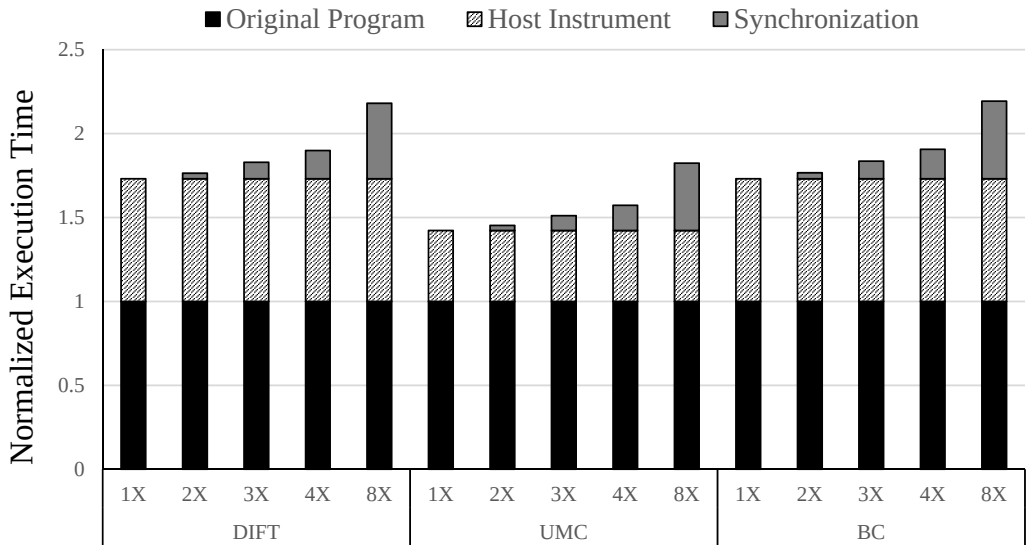


Figure 2.14 Execution time of PAUD when PAU is paired with higher frequency host processor(normalized to native)

In Section 6, as a practical example where the programmability of TPC can be utilized, we introduced the multi-level tracking optimizations for DIFT. In Figure 2.15, the performance improvement achieved by the optimizations is shown, for the eight applications of mibench. The configuration PAUD_multi is the optimized DIFT implementation explained in Section 6. For the fair comparison, we also apply the optimizations to other approaches. In the two

configurations, SWD_multi and MPD_multi, the same optimizations are added to SWD and MPD respectively. The performance improvement of the block-level optimization is affected by the taintness of the input [73]. To maximize the performance improvement, we assume that the input files accessed by the applications are not the confidential ones so that the tag for the input is not set. Thus, virtually all basic blocks are skipped by the block-level optimization although the computations for the decision are still required.

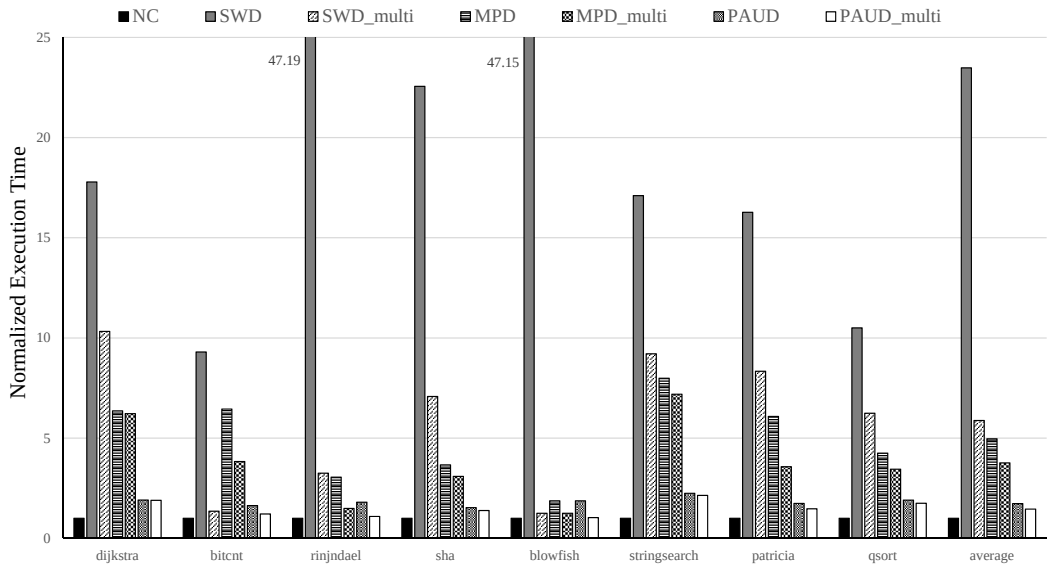


Figure 2.15 Comparison of execution time for DIFT implementations (normalized to native)

As depicted in Figure 2.15, the adoption of the optimizations improves the performance in all the approaches compared in our experiment. SWD_multi is about 4 times faster than SWD, and MPD_multi improves the performance of MPD by 31.9%. In our approach with PAU, the performance of PAUD is also enhanced by 18.8% with the optimizations and it consequently reduces the

DIFT overhead to only 45.7% in comparison with NC. In Figure 2.16 shows the execution time of PAUD_multi for various performance gap ratio between the host and PAU. As shown in the figure, the optimizations applied to our DIFT implementation can reduce the DIFT overhead substantially. Even when the performance ratio is 8x, the increased execution time is about 33.1%. The results show that the programmability of PAU can help our DIFT implementation to improve the performance by taking the advantage of software's flexibility and to be more tolerable to the performance ratio.

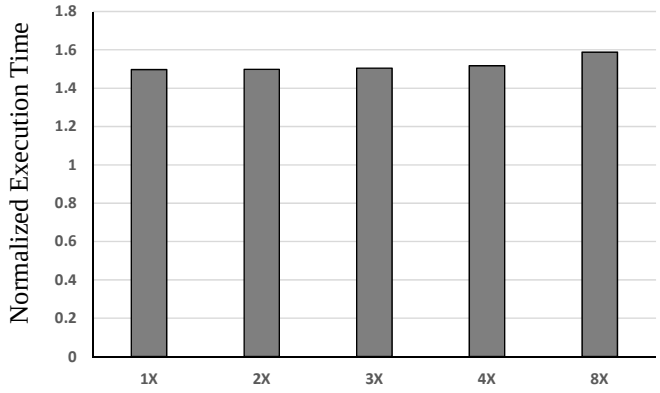


Figure 2.16 Performance overhead of PAUD_multi when PAU is paired with higher frequency host processor(normalized to native)

2.8 Related Works

Most software DPA approaches [20,29,62,73,79,82,99,104] have relied on binary-code instrumentation to augment the codes for DPA to carry out their analysis schemes. However, even in simple analysis such as array bound checking, it requires substantial analysis computations [91]. For example, Memcheck [82] uses dataflow tracking to detect a wide range of memory errors in programs as they run. Under the analysis, the monitored program typically run 20-30 times

slower than normal. Although it is paid at test-time, the performance overhead sometimes limits the amount of analysis due to restricted time for the software development, thereby making it difficult to remove all errors in the program. In case of DIFT, the performance overhead of software-based solutions [20, 62, 99] reaches up to 37 times the original code execution [62]. Several efforts were made to curtail the overhead with optimization techniques [73,104], but it yet remains one or two orders of magnitude higher than the execution time of the original program. Considering that DIFT is usually used for runtime monitoring, the analysis performance is not acceptable level to be deployed in real applications.

In order to improve the analysis performance, there are several software-based approaches to utilize multiprocessors [19,59,63] that are readily available in modern multicore architecture, such as Intel i7, where each core is a GPP. The key idea here is to devote GPP cores to run *helper threads* whose missions are actual analysis for the host program running concurrently on another GPP core. For example, Speck [63] offers up to 7.5X speedup with 8 cores for light-weight analyses like scanning the address space for sensitive data. However, although they can lessen the performance overhead with existing architectures, the achieved performance is not sufficient for more powerful analyses, mainly because the original GPP architecture is not optimized for program analysis in the first place [27]. For instance, in [18, 19, 59], the program execution times get 3-7 times slower when the analyses being enabled so that it is too slow to be used for runtime monitoring. Moreover, even in test-time analyses, there is also a demand for more complex analysis tools that incur overheads from 100 to 300 X [58, 81, 91, 95].

To mitigate the performance overhead, in several multiprocessor approaches [19, 59], they modified the host CPU's internal architecture and integrated the specialized hardware modules like our trace buffer. By doing so, they were able to

reduce the overhead to acceptable levels, which is around 50% or less [59] in DIFT problem. Nevertheless, such modifications in these approaches also impose the same problem of the core-level approach that mandates the alteration of existing commodity processors.

To address the shortcoming of software-based analysis, several core-level hardware engines have been proposed [19, 22, 23, 27–29, 43, 56, 92, 96, 102]. In those approaches, extra hardware logics customized for analysis operations are integrated into a processor. A number of DPA schemes are supported in the core-level engine [27] such as fine-grained memory protection [96], array bound checking [29], software debugging support [102], managed language support like garbage collection [43]. The main advantage of the core-level approaches is that they do not need to instrument the host code since they can extract the necessary information from the processor's pipeline transparently. Thus, they could bring the overhead down to under 5%. However, they have a disadvantage in that invasive modifications to the processor internal (e.g., registers and pipeline data paths) are required. In fact, modern microprocessor development may take several years and hundreds of engineers from an initial design to production [27, 44]. Therefore, the substantial costs of development to integrate the customized logic would hamper processor vendors to adopt them, unless the necessity is clearly established.

Several previous works [27, 28, 30] have been proposed to leverage the flexibility to support various DPA schemes, generalizing from the core-level engines. FlexCore [28] is a hybrid architecture that combines a general core with a decoupled on-chip FPGA fabric. Although the FPGA logic can be reconfigured to conduct a set of DPA schemes in hardware, the low throughput of FPGA can cause high performance overheads [27]. To mend this problem, in Harmoni [27], they proposed a high performance and reconfigurable co-processor for a wide

range of DPAs. With the considerations on the tag-based DPA model, Harmoni has the specialized pipeline architecture which can achieve very high performance, while it also has sufficient flexibility thanks to the configurable tables in the engine. Nevertheless, as discussed in Section 1, it still has the extendibility issues when a new analysis method is suggested.

In recent years, for DPA techniques or malware detection, the system-level hardware engines like our PAU have been proposed, which do not require any modification on the host core [48, 57, 69, 91]. Among them, Hardgrind [91] is the previous work closest to ours where the computation of an instrumented program is paralleled between the host and the accelerator. However, since they only quantified the potential of this approach by considering how the execution traces are delivered to the engine, the detailed structure of the accelerator was not sufficiently addressed. On the contrary, in this chapter, we designed the detailed architecture of our PAU that supports various DPA schemes as well as enhances the analysis performance, in order to leverage the system-level approach.

Another difference between Hardgrind and ours is the method for the trace communication. In Hardgrind [91], they allocate a buffer space in the host memory and store the traces into the region. Once the buffer is full, DMA transfer is triggered so that the traces are delivered to the analysis engine in a bulk. As compared to our approach with the trace buffer, when this transfer method is used, the buffer access latency can be reduced because the buffer that can be cached by the host CPU's cache is much faster than the trace buffer located in the off-core module. On the contrary, the DMA-based transfer mode increases the number of the added instructions to manage the buffer in the memory. That is, there exists a trade-off between the two overhead sources; the number of instructions and the access latency to the buffer. In our work,

we chose to use the trace buffer because the access latency of the trace buffer located on the system bus is relatively short and it is more efficient than the DMA-based transfer mode in our prototype. However, in desktop platforms where PAU is implemented as a PCI card, the DMA transfer mode might be more efficient way to communicate, as presented in Hardgrind [91].

As one of complementary works for our PAU, it is noteworthy that Log-Based Architectures (LBA) [18,19] proposed a decoupling execution strategy in the context of the multiprocessor approaches while it also have core-level logics to compress, deliver and decompress the traces of host programs. By employing the LBA architecture, the host can deliver the traces to our engine without instrumenting the host code thereby improving the analysis performance. Although this architecture is not yet available in the commodity market, we hope that it will be implemented and sold as a commercial product in the near future.

In several hardware engines [44,54,84], they have proposed to utilize special channels for acquiring runtime information, without the modification on the host CPU's internal pipeline. Although they had to slightly modify the hardware design of host CPU to provide such channels in their works, it is noteworthy that this problem can be resolved by incorporating the trace interfaces available in recent commodity cores. For example, the recent ARM processors, such as Cortex-A9 or A15, include the CoreSight architecture [6] to support efficient and convenient tracing. It can provide the analysis engines with various runtime information such as branch results, context switches and exceptions, without incurring performance overhead for trace communication. If the interface can be combined with the hardware engines, they can achieve high performance in DPA computations while the system-level integration is still viable. In this context, it is noteworthy that in Extrax [49] proposed by J.Lee et. al, the core debug interface available in many CPU architectures is employed for efficient kernel

integrity monitoring. Since the interface can provide many informative signals to retrieve the context of runtime execution without performance loss, Extrax can detect any malicious attempt to compromise the kernel with negligible overhead. Although they only focused on the kernel integrity, we believe that the use of the core debug interface can be exploited in DPA. Motivated from this, we also have a plan to incorporate the interfaces to PAU in our future work, in order to achieve both the programmability and performance improvement.

2.9 Chapter Summary

This chapter presented a system-level hardware engine, called PAU, which is an application-specific programmable processor to support a wide range of DPA techniques with the enhanced analysis performance. PAU can speed up the analysis performance with the help of specialized architecture based on the tag-based model, which otherwise would be substantially slow as in computations on GPP cores. In addition, with its programmability, it can support a wide range of DPA techniques and enable flexible computations for evolutionary analysis strategies. In our case studies, we demonstrated the effectiveness of our approach by realizing several DPA techniques on our PAU and successfully adopting the software-assisted optimizations for DIFT. Moreover, following the system-level approach in Hardgrind, our PAU has been designed as a system-level component without any modifications in the host processor internal and it is integrated with an existing platform. Therefore, our approach can be easily implanted to a commercial mobile platforms or desktop ones.

Our experiments on FPGA prototype revealed that our solution can reduce the DPA performance overhead substantially compared to the previous solutions. While multiprocessor approaches slow down the execution of a program

by more than a factor of 4, our PAU incurs overwhelmingly low overhead, that is only 45.7% for a group of mibench applications in our DIFT implementation. Even when our PAU is several times slower than the host processor, the DIFT overhead increases only slightly about 33.1% for the same applications. The experiments also revealed that the power consumption and area overhead of PAU are acceptably small compared to today's mobile processors. All in all, we hope that our proposed ASIP approach would become an attractive DPA solution to production-quality commodity platforms.

Chapter 3

A Practical Solution to Detect Code Reuse Attacks on ARM Mobile Devices using an On-chip Debug Module

3.1 Introduction

Since ARM released its first processor architecture in 1985, it has developed a large number of processors which have lower power consumption, yet high performance. Today, ARM processors are undoubtedly deemed as the de-facto standard CPUs for diverse smart mobile devices including smartphones and tablet PCs. As smart mobile devices continue to gain in popularity among the general public for everyday communication and information processing, they are becoming more appealing targets of numerous software-oriented attacks in recent years. The ultimate objective of these attacks is mostly to possess the capabilities which empower them to control the system behavior in almost all aspects so that they can capture various system events and react to the events

for their profits.

A popular method to acquire such formidable capabilities has been *code injection*; that is, attackers first inject their own code in the memory and forcefully execute the code after hijacking the normal course of execution [70, 94]. The most effective measure against code injection would be to eradicate the possibility of unauthorized code injection and/or injected code execution in the first place. To regulate code injection/execution in the system, modern processors support non-writable and non-executable page permissions with which the OS kernel can enforce the *Writable xor eXecutable* (W⊕X) policy. Although the policy has been proven effective enough to prevent attackers from hijacking control flows of user applications via code injection [88], there has been more recently emerged a new breed of attacks, called *code reuse attacks* (CRAs), in order to neutralize the protection under the policy. Technically, these attacks obey the W⊕X rule since they rely not on injected code but on existing legitimate code in the victim machine. To launch a CRA, the attacker analyzes the target programs and collects a set of code snippets, called *gadgets*, from existing code blocks. By stitching gadgets into a new code sequence, the attacker can perform Turing-complete computation without injecting any additional code. By exploiting common buffer overflows, CRAs can be crafted to target virtually all modern machines like such smart mobile devices as have been targeted by diverse CRA schemes primarily for jailbreaking [86].

With the CRA threat being more significant, the CRA problem has been addressed by various solutions [17, 21, 25, 45, 46, 68], which come in various forms of either software or hardware. The clear advantage of software solutions is that they can be easily adapted to the present machine platform. Their drawback, however, is that they may impose tremendous computational loads upon the host machine mainly because the original program must be augmented

with extra code that will be executed periodically to check abnormal control transfers on the host during runtime [17, 68]. Obviously, such a considerable amount of computational overhead can be the biggest obstacle that would impede software solutions from being widely deployed in smart mobile systems that often suffer from severe limited resources. On the other hand, the hardware solutions [25, 45, 46] have demonstrated their strength in performance because they can excel at CRA detection with assistance of special hardware logics customized for this task. To maximize the performance, these solutions all require intrusive modifications to the original CPU internal architecture in a way that their special hardware can be tightly coupled within the host CPU for close monitoring of every control transfer during code execution. Despite their dramatic performance enhancement, fulfilling this requirement however would stymie their direct deployment into existing ARM mobile devices. The reason is that it is contradictory to a common design practice for an ARM device in industry today. In each device lies an *application processor* (AP) [64, 74, 78] as the central computing platform for applications running on the device. To meet ever increasing demands for low design cost, high performance and fast time-to-market, device vendors these days usually build the AP platforms for their products by buying one of ARM cores off the shelf and integrating it together with supporting IPs (intellectual properties) optimized for specific functions. However, if they want to adopt some of those CRA hardware solutions for their products, they cannot follow this usual convention in the hardware design of an AP platform. Instead, they will be compelled to spend their time modifying the ARM core architecture itself including the registers and pipeline datapaths, contrary to the general convention.

Our observation on earlier work inspired us to develop a more practical hardware solution that is to facilitate the acceptance of hardware technologies

for the CRA problem in today's smart devices with ARM-based APs. In our solution, all hardware IPs for CRA detection are placed outside the ARM CPU and connected together with this host CPU to build the target AP platform, complying with the conventional design principle. In this chapter, we introduce our preliminary architectural design for an ARM-based AP where we have integrated the hardware IPs to detect a representative technique for CRAs, called the *return-oriented programming* (ROP). The ROP attack, as the name implies, chains the gadgets, each of which ends with a return instruction, by manipulating their return addresses on the stack through the exploits of buffer overflow vulnerabilities. Our hardware IPs are basically exerting the same strategy for ROP detection, called the *shadow stack*, that have been employed by earlier work [26, 67]. However, the main difference is of course that our hardware is to monitor ROP attacks from outside the CPU while theirs were designed to watch the attacks from the inside.

The real challenge here for our approach is how we overcome the limited visibility into the CPU inside so that our monitoring hardware can secure the same quality of information about the host code execution as being readily available to those internal monitors directly from abundant resources within the CPU. To tackle this challenge, we must somehow provide our monitor with a special mechanism that can expose all necessary host code execution information for CRA detection outwardly in a timely manner. Luckily, we have found that ARM is already equipped with a special architecture, called *CoreSight* [6], that corresponds roughly to this mechanism. CoreSight, available in virtually all ARM processors including Cortex-A8, A9 and A15, has been originally developed to supply the outside devices with the information about real-time debugging and tracing of running code in the host. To utilize this architecture, the devices should be attached to the ARM debug interface, such as the *trace*

port interface unit (TPIU) and *program trace macrocell* (PTM), from which they can obtain in real time a trace of branch outcomes produced during code execution. In our target AP, for our security purpose, we have attached the ROP monitoring module to the host ARM processor via CoreSight TPIU so that our monitor can see every branch trace of a potential victim program and catch any suspicious call/jump patterns that may indicate CRAs. However, being devoted to its original design purpose, this ARM debug interface carries the minimum information about branch behaviors necessary to keep track of program execution flows for debugging. For instance from CoreSight, we only obtain two kinds of branch outcomes: the target address of an indirect branch and the direction (taken/not-taken) of a direct branch. Sadly for our purpose, these are not sufficient enough; for its accurate monitoring task, our hardware needs to distinguish the differences among various branch types such as direct/indirect calls, returns and other direct/indirect jumps. To supplement this lacking information for our modules, we perform the offline binary analysis for each program and generate the *meta-data* that will direct at runtime the external modules how to obtain the exact type for every branch in the traces from CoreSight TPIU.

The rest of the chapter is organized as follows. Section 2 describes the previous studies related to ours and also the threat model with our assumptions. After Section 3 presents the overall hardware architecture of our ROP monitoring module, Section 4 explains in detail how ROP attacks are efficiently detected with help of additional code analysis in our approach. Then, Section 5 discusses the experimental setup and results. For the setup, we have used an ARM-based Zynq FPGA board [32] and prototyped our hardware modules to build a full AP system on the board. The results show that our prototype system offers a feasible security solution for protecting ARM-based APs against

ROP attacks with high speed and low area overhead. Finally in Section 6, we summarize this chapterfr.

3.2 Related Work and Assumptions

In this section, we first relate our work in more detail with others. We then define the threat model assumed in this work.

3.2.1 Related Work

In [67], as one of the hardware solutions for CRA detection, they propose a hardware solution called SmashGuard which protects the host system against ROP attacks. In their approach, on each function call, return addresses are saved in a hardware stack added to the CPU. A return instruction pops the most recent return address from the top of the hardware stack, and then the popped address is compared to the return address of the program at runtime. If there is a mismatch between the two addresses, it is highly likely that the return addresses are maliciously manipulated by attackers. More recently, the branch regulation technique to detect CRAs is introduced in [45] on the ground of a simple invariant ruling the normal behaviors of branches in a programming language. The invariant rule says that the target of a branch instruction should point to either the address of a function entry or an address within the same function that the instruction belongs to. To enforce this rule, they first rewrite the original binary to annotate each function entry in the victim code with the information delimiting the function boundaries. Then during code execution, their special hardware checks if any branch violates the rule. Being installed within the host CPU pipelines, the augmented hardware was able to efficiently monitor every branch behavior in a timely fashion. SCRAP [46] is another no-

ticeable hardware-assisted technique aiming to detect CRAs based on unique signatures that characterize the execution patterns of instructions commonly encountered when the CPU is under the attacks. As in the case of branch regulation, their hardware is implanted inside the CPU, more specifically at the commit stage of the pipeline. As another related work, the researchers in [25] use their hardware to confine indirect calls and returns only to target the valid addresses, and apply software heuristics to pinpoint CRAs by analyzing their execution patterns. Especially for a function return, they use *active labels* to ensure that the function returns to another function that is currently active, that is, not returned yet after being invoked. These hardware studies for the CRA problem empirically suggest that capitalizing on hardware techniques should be an excellent way to conquer the performance issues, inherent to this problem, from which pure software solutions often suffer significantly. Unlike ours, however, their techniques have difficulty in being directly deployed in most modern smart devices with ARM-based APs, as discussed earlier.

Similar to our effort of extracting accurate runtime information inside the target system, others also have made attempts to use the debug interface like ARM CoreSight. In [34], the On-Chip Debug infrastructures were used to test the fault-tolerance of the target system. Through the interface, they access internal resources including registers and memory so as to inject faults into the resources of interest and analyze the system response in a non-intrusive manner. In [72], researchers proposed an on-line fault detection technology by reusing available debugging features of existing processors like LEON3 and ARM7TDMI. Although all these studies and ours both are commonly exploiting the built-in debug interfaces, they use the interfaces for just the fault detection techniques, not the security-enhanced system that we do for.

Very recently, to the best of our knowledge, there is the first approach [49]

that utilizes the debug interface in an effort to thwart security threats by developing a kernel integrity monitor. To detect any attempt to compromise the kernel in the target system, the monitor incessantly snoops the memory traffic to track all alterations made to the memory regions for critical kernel data. In the original design, the monitor was to watch the memory bus for all write transactions issued from the host CPU to the main memory so that it could capture malicious transactions towards the kernel regions as soon as they appear on the bus. However in reality, exploiting the memory hierarchy, attackers may deliberately hold their altered data in the on-chip cache before flushing it onto the bus, long enough to hide their attacks from the monitor. Via the debug interface, the monitor came to successfully extract the cache resident information and consequently uncover some of the attacks.

3.2.2 Threat Model and Assumptions

We make the same assumptions on CRA that had appeared in previous studies [21,26,46,68]. We first assume that by enforcing the W$\oplus$X security protection rule [2,7], the OS and CPU cooperate to forbid a memory page from being both writable and executable at the same time, and subsequently that adversaries cannot execute their injected code. Under this assumption, to disable the defense mechanism and do additional attacks, the adversaries must gain sufficient privileges for the first time. We hereby assume that there are no other attack vectors or security holes which can directly escalate adversary's privilege.

As another assumption, adversaries might exploit memory corruption vulnerabilities like buffer overflows to hijack the control flow of the victim software by overwriting control data in the stack or heap. After they gain the control flow, they have to execute complex operations (e.g., privilege escalation, executing files, and reverse connection) using CRA techniques. Also, we do not

consider the adversary who intends other arbitrary attacks such as denial of service. All underlying system operations that are performed by the OS kernel and hardware are always secure until they are thwarted by the sufficient privilege of an adversary obtained through CRAs.

We also assume that an adversary knows all implementation details of the target application and locates the exact address of gadgets. Although *address space layout randomization* (ASLR) [89] can prevent the adversary from locating the address of gadgets, it is here assumed that the adversary still can bypass ASLR and reveal the memory layout by exploiting memory leak bugs like a format string bug [61]. In addition, we assume that the application is not compiled by attackers so that a number of useful gadgets cannot be contained within a small code base. Lastly, the self-modifying code is not considered in our assumptions because it conflicts with the W⊕X security protection.

3.3 Architecture for ROP Detection

As clearly stated in Section 1, the ultimate goal of our research is to build a practical hardware solution for CRAs that is deployable to modern ARM-based AP platforms. As the first step towards this goal, we have implemented and integrated monitoring modules for ROP detection as a subsystem, called the *ROP monitor*, in an AP platform with an ARM CPU. Figure 3.1 displays the overall design of our hardware built on top of the platform where as an off-core hardware IP, the monitor is assembled together with the host CPU as well as other IPs. As shown in this figure, we expect that our security subsystem can be readily incorporated into modern mobile products with ARM-based AP platforms. In our platform, the CPU is an ARM Cortex-A9 processor [9] which has been installed in a large number of commercial devices these days [35, 76].

Also, observing the convention of AP design, the host CPU and our hardware are connected with the shared main memory via the standard AMBA3 AXI interconnect like ARM NIC-301 [8]. It is noteworthy that we have strived to design all our modules to comply with the standard protocols and specifications of commercial ARM-based AP platforms in the present. Specifically, the modules communicate with the ARM CPU, conforming to the AMBA3 bus protocol and the CoreSight debug interface specification. In accordance to our proposed approach, the branch traces should be emitted from the host, and transmitted to the ROP monitor via the ARM CoreSight PTM and TPIU.

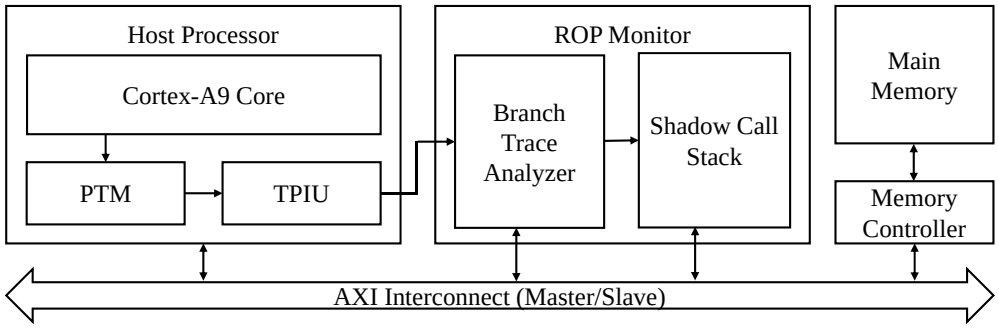


Figure 3.1 Overall architecture of our AP design

As depicted in Figure 3.1, the ROP monitor is largely divided into two modules: the *shadow call stack* (SCS) and *branch trace analyzer* (BTA). In our AP design, SCS plays the pivotal role of monitoring ROP. Upon receiving all branch execution patterns possibly relevant to ROP attacks, SCS analyzes the patterns and judges whether or not the patterns indeed result from the attacks. BTA is an additional module that connects SCS to the TPIU debug interface of the host CPU. Its central role is to decode the debug information from TPIU and to properly refine the information for analysis before the delivery to SCS. Below we will give the detailed descriptions of aforementioned hardware

modules of our proposed design.

3.3.1 Branch Trace Analyzer

As discussed before, ARM CoreSight has been used by many developers to debug or evaluate their software on the devices. PTM is the key module of CoreSight that captures diverse debug information of the ARM CPU, such as branch target addresses, exceptions, current process IDs and instruction set mode changes (ARM/THUMB). It also produces the generic form of the tracing data and compresses the data according to the CoreSight program flow trace architecture specification [4]. After compression, the generated PTM traces are routed to TPIU, and finally forwarded to the off-chip pins to provide the external modules with the runtime information of host programs.

In the current implementation (Figure 3.1), the output signals of TPIU are directly routed to the on-chip ports of the ROP monitor instead of the off-chip pins so that we can utilize the CoreSight modules within our AP prototype without leaking any internal information to outside the AP. We have also built a device driver running on the Linux kernel to control the functionalities of PTM and TPIU. As soon as the CoreSight modules are activated, the PTM traces are generated and transferred to BTA in our ROP monitor via TPIU. In our prototype, PTM is configured to generate a trace for every branch that indicates whether the branch is taken or not. If the branch is indirect[1], then the trace also includes the branch target address.

Figure 3.2 depicts the internal structure of BTA. A main submodule in BTA is the *trace analyzer* that decodes the PTM traces to extract the branch type[2] and target address of each branch instruction executed on the host. The

[1] that is, an indirect call, indirect jump or return

[2] There are five types: direct jump/call, indirect jump/call and return

concern here is that the host CPU generally operates faster than other supportive IPs including our monitor, and as a result, the trace analyzer may not process instantly all the traces arriving from the host. To resolve this, we provide an asynchronous memory queue, called the *branch trace FIFO*, in order to temporarily store the incoming traces for the analyzer.

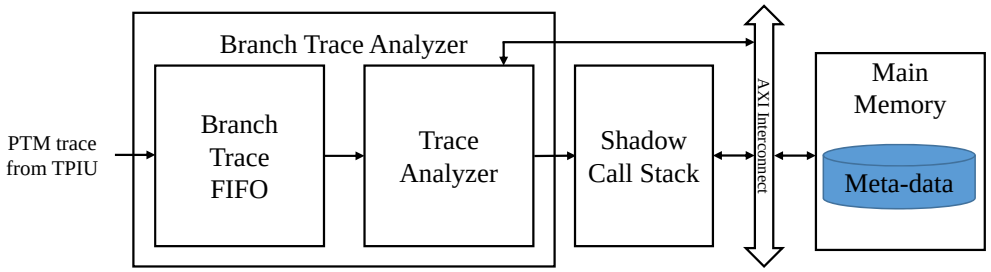


Figure 3.2 Hardware architecture of BTA

In most cases, the traces alone do not give sufficient information for the trace analyzer to correctly interpret the current branch behaviors in the host CPU, which is an indispensable step to reveal the existence of a ROP attack in our system. For such accurate trace analysis, our SCS in principle needs to know three pieces of the information regarding the behaviors: the branch target addresses and branch types and branch outcomes (taken/not-taken). Unfortunately, the traces coming through TPIU do not disclose the branch types. To supplement this information, we perform the offline binary analysis before running a program, during which we generate the set of meta-data for the type of every branch and store them in the main memory. Combining the meta-data set with the traces coming from TPIU will constitute the complete information about branch behaviors within the CPU that is necessary for our monitor to analyze the presence of ROP attacks in the system. During code execution, as soon as identifying the type of each instruction, the trace analyzer

determines which information should be extracted from the incoming traces. The extracted information is then sequentially fed as a trace of CPU execution into SCS which keeps track of all occurrences of branches in the trace for ROP detection.

3.3.2 Shadow Call Stack

When attackers try to compromise a system with ROP attacks, they manipulate the return addresses stored in the stack in a way to assign them arbitrary values different from those initially set by the callers. In [26, 67], the idea of shadow stacks has been proposed to detect ROP attacks by capitalizing on their distinctive characteristic. Adopting this idea, we have designed SCS to make a copy of the return address on every call instruction and to match the copies with the return addresses coming from BTA.

Figure 3.3 shows the hardware architecture of SCS. Note that SCS have three input signals flowing from BTA; one is `addr_in` for the return address of a branch instruction, and the other two are `call` and `return` for branch types. Using `call` and `return`, the *queue controller* composes the three signals `push`, `pop` and `counter`, and forwards them to the *address queue*, whose job is to maintain a shadow copy of the call stack on the host side. When a function is called, the controller sets `push` on and stores the value of `addr_in` into its queue. Then at the next cycle, it increases the counter value by one. When the function returns, the controller sets `pop` on, outputs the front queue value through the `addr_out_queue` line, and decreases the counter value by one.

Note that the address queue has a finite number of entries, eight in our work. Due to this limited number of its entries, the queue will be overflowed if the victim code contains more than eight times nested calls. To cope with this exceptional case, we reserve a memory region, called the *SCS region*, in the

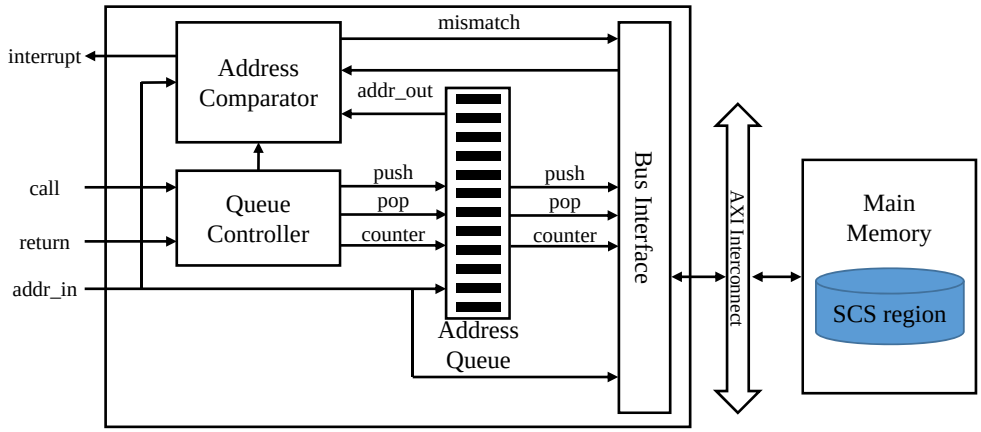


Figure 3.3 Hardware architecture of SCS

main memory. When the queue is full, its controller saves into the SCS region the values of `addr_in` for all subsequent function calls. Later when one of these functions returns to its caller, the controller fetches the saved return address from the SCS region and outputs it via the `addr_out_mem` line.

Figure 3.3 shows a multiplexer that is connected to two input lines, `addr_out_queue` and `addr_out_mem`, respectively from the different sources for saved return addresses. The multiplexer chooses from which source a return address is transferred to the output line `addr_out`. The choice depends on the value of `counter`. For the value ≤ 8, the queue should be the source. Otherwise, the other source will be chosen. The destination of `addr_out` is the *address comparator* that is in charge of making a final decision about the existence of an ROP attack. For this, the comparator takes two input values from `addr_in` and `addr_out`. The former value represents the actual address that the current function is about to return, and the latter the original return address saved when the function was called. If these values do not match, the comparator interrupts the host CPU to notify of an ROP attack. To summarize, in our design, the value of `addr_in`

is pushed into either the address queue or SCS region at a call instruction, and compared at a return instruction with the value of `addr_out` from the address queue.

3.4 Meta-data Construction

As discussed in the previous section, the meta-data plays a pivotal role for the success of our ROP detection mechanism built on top of the monitoring hardware platform. In this section, we first describe how the data is generated from a target program, and then the mechanism that makes its use to detect ROP attacks.

Figure 3.4 illustrates the ROP detection process using our ROP monitor. It basically consists of two phases; the static binary analysis and the runtime detection process. In the analysis phase, users generate all the static information of their target programs necessary for ROP detection schemes. For this, they are provided with the *binary analyzer* that is to find the information regarding branch behaviors in the program binaries. As explained earlier, BTA and SCS both are particularly interested in the information related to indirect branch instructions such as returns, calls, indirect jumps and functions' boundaries. With the help of the analyzer, the users can extract this information from their programs. After the binary analysis, the resulting information is summarized in the form of meta-data whose objective is helping the ROP monitor to better understand the execution behaviors of the target program running on the host and to check if there is any behavior possibly related to ROP attacks. At runtime, when the program binary is loaded by the OS kernel into the host, the associated meta-data for the detection process is also downloaded to the pre-defined region in the main memory. The data can be accessed by BTA to serve its needs. In the following, we will describe the meta-data structure and

how the data are used by BTA to detect ROP attacks.

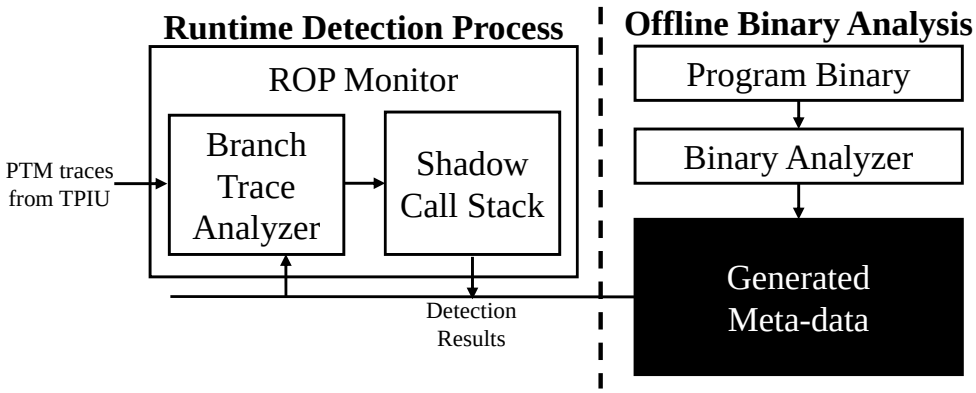


Figure 3.4 ROP detection process

3.4.1 Meta-data Structure

To understand how the ROP monitor works with meta-data, we briefly explain the relevant aspects of the ARM processor architecture. The 32-bit ARM processor is provided with 16 general-purpose registers (`r0-r15`). A key difference between the x86 and ARM architectures is that all ARM registers can be accessed directly by ordinary instructions. Even the program counter (PC) is aliased to `r15`, and thus can be modified by various types of instructions including moves, arithmetics or loads/stores if it is their destination register. As a result, the control flow of a program can be changed by not only branch/jump instructions but also other types of instructions.

The function calling convention is described in the document for the ARM architecture procedure call standard [52]. According to the document, function calls are implemented by `bl` (branch with link) or `blx` (branch with link and exchange) instructions. Both instructions perform a branch with the link operation that changes PC while the return address is saved to the link register (LR), which is aliased to `r14`.

In the ARM architecture, any instruction that may change PC can be used as a return. The most common method is to use a `bx` (branch exchange) instruction with LR. The instruction `bx lr` replaces PC with the saved return address in LR (`r14`). Another way is to use the `ldm` (load multiple) or `pop` instructions that take PC as the destination operand. In this case, the return address which was pushed on the stack is restored to PC. In the case of indirect branches, the `bx` instruction is executed with the register operand storing the target address.

Following the convention for calls, returns and indirect branches, the binary analyzer extracts the branch type for each branch instruction of the program binary. The extracted branch types are stored in the main memory as meta-data. In addition, to support SCS, BTA should deliver the return and target addresses for branch instructions. Recalling that only the target address of an indirect branch and the direction of a direct branch can be acquired from the traces coming through TPIU, BTA has to generate the target addresses of direct branches and the return addresses for all types of branch instructions. For this purpose, we keep the types and the source and target addresses of branch instructions in the form of meta-data.

As the first step of binary analysis, the analyzer divides the application code into multiple code regions. For this, it scans the entire code from top to bottom. First, it creates a new code region starting from the entry of the application. Now, scanning down the code, it constructs the region by including in order every instruction following the starting point until it hits a control transfer instruction that is either a branch (direct/indirect) or a return. This control transfer instruction will be the last instruction to be added to the current code region. Upon completing the construction of one code region, the analyzer initializes a new region beginning with the instruction immediately next to the last region, and repeats the above procedure for this region with the instructions

in the remaining application code.

In Figure 3.5, we show an example of code regions created by the binary analyzer. Given the resulting code regions, the analyzer generates two types of meta-data for the application code: the *region info set* and *region info jump table*. The former contains the source and target addresses of every control transfer instruction, which is positioned at the end of a code region. The region info jump table is composed of entries each of which maps a code region onto the region info set. By using this table, we can access the information about code regions in the set. The table is located in the main memory with a predefined offset relative to the address where the target application is loaded. The region info jump table has an additional field, *B-type*. The B-type tells the branch type of the last instruction in the current code region.

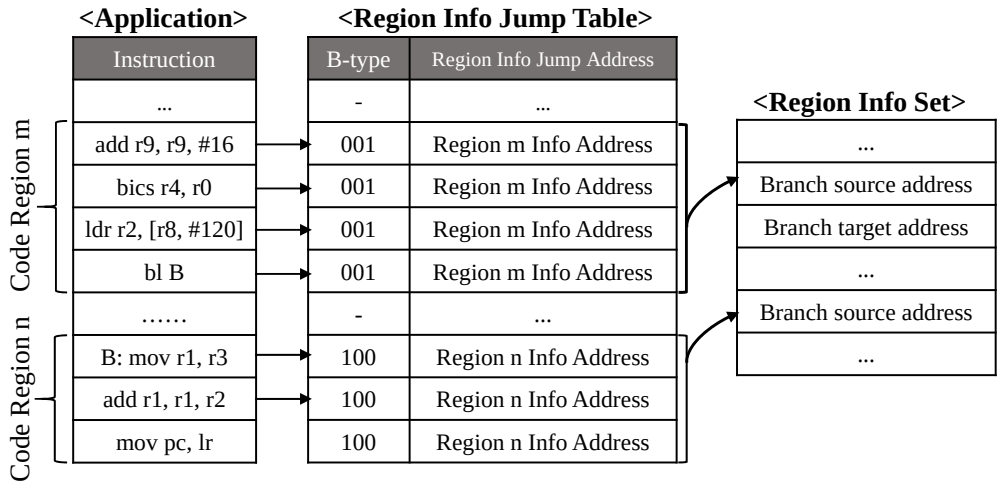


Figure 3.5 Meta-data layout for the ROP monitor

Table 3.1 shows the contents of the region info set where we can see that the set contains different information for different branches depending on their types. For instance, for a direct call instruction, we need both its source and target addresses in the meta-data. Recall that the traces from the TPIU interface

do not carry the target addresses of branch instructions. Therefore, the ROP monitor should obtain the address information from the meta-data. The source address is also necessary to calculate the return address (= source address + 0x4) which then will be delivered to SCS. On the other hand, for an indirect call, the monitor only needs the source address because it can glean the target address from the incoming traces from TPIU. In the following subsection, we will present an example of meta-data and show how each submodule of our ROP monitor obtains the necessary information from the data.

Value	Branch Type	Branch Information
000	direct jump	source/target address
001	direct call	source/target address
010	indirect jump	source address
011	indirect call	source address
100	return	source address

Table 3.1 Information for different branch types

3.4.2 Using Meta-data for ROP Monitoring

An example of meta-data is depicted in Figure 3.6. Let us assume that the control flow has reached the address 0x8040 in the example. The instruction at this address is within the code region 0 whose last instruction is a direct call, `"bl func_3"`. As a direct call requires both the source and target addresses, BTA reads these addresses from the address `A` which points to the information of the code region 0 in the region info set. In the example, the return address of the direct call is 0x8048 (= 0x8044 + 0x4). BTA sends the calculated return address to SCS and sets the `call` signal on at the same time. When the `call` signal is on, SCS pushes the address coming from BTA into the shadow stack. The function `func_3` has its return instruction `"mov pc, lr"` at 0x8074. When this instruction is conducted, BTA delivers the target address coming from TPIU, which is the return address, to SCS and simultaneously sets the `return` signal

on. When the `return` signal comes to be on, SCS is notified to pop from its shadow stack. Then, the stack top value is compared with the delivered return address to determine the legitimacy of the return instruction. Let's assume `r2` points to the entry of an arbitrary function such as `func_4` when the control reaches the instruction at 0x8050. Because the instruction is an indirect call, only the branch source address is provided as meta-data for this instruction. But as the return address of a call should always be `source address` + 0x4, integrity checking procedure remains the same as that of a direct call. Following the above procedure, the return address for every call is pushed into the shadow stack to be compared later with the actual return address when the control reaches a succeeding return.

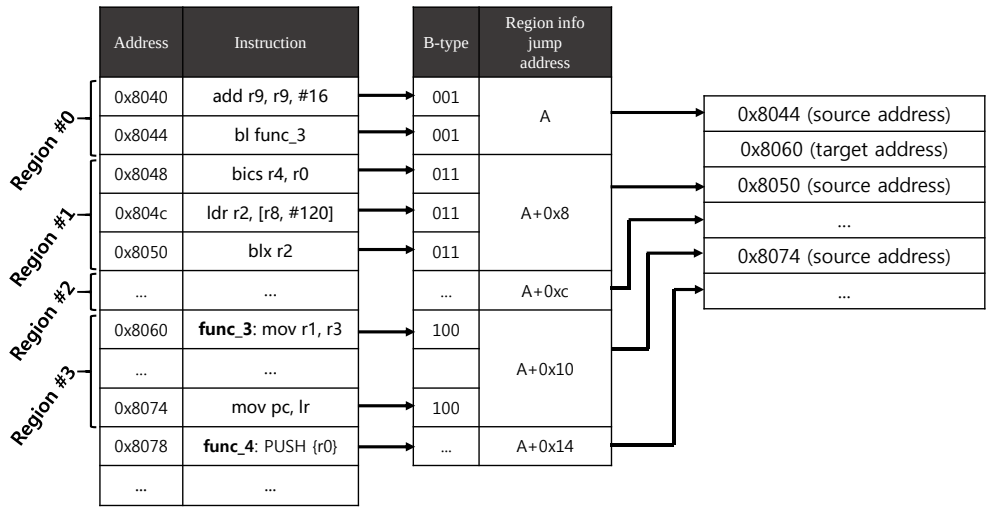


Figure 3.6 Example of meta-data

3.5 Experimental Result

To evaluate our approach, we have implemented a full-system SoC prototype on the Xilinx ZC 702 evaluation board [32]. This development board is composed

of the Zynq-7000 XC7Z020 platform which is equipped with a dual-core ARM Cortex-A9 processor, ARM NIC-301 AXI interconnect, an FPGA chip with 1.3 million gates, 1GB DDR3 SDRAM and other peripherals. We have built the host system with the A9 processor and deployed Xilinx ARM Linux kernel version 3.8 as the host OS. Also, two CoreSight modules, PTM and TPIU, in the Cortex-A9 processor are enabled so that we can extract branch traces from the host CPU. The operating frequencies for our ROP monitor and the host CPU are 90 MHz and 200MHz, respectively. To support the asynchronous relation between the clocks, there is an asynchronous bridge in PTM. The ROP monitor also contains the branch trace FIFO of 16 entries in BTA. Based on the parameters for the prototype mentioned above, we have synthesized the ROP monitor onto the FPGA chip in the evaluation board and quantified the logics necessary for the hardware modules of the ROP monitor including BTA and SCS in terms of lookup tables for logic (LUTs) and memory elements (BRAMs). The synthesis result shows that our ROP monitor occupies 13.8% (7,362/53,200) of total LUTs and 3.1% (539/17,400) of total memory elements. To complement the result, we also measured the gate count of our ROP monitor using Synopsys Design Compiler. With a commercial 45nm process library, the gate-count of the proposed monitor is 86,714. Considering that the ARM Cortex-A9 dual core requires about 26 million gates [65], the area overhead for adopting the ROP monitor into the emerging AP platforms seems to be acceptably small.

To measure the performance overhead of our ROP monitor, we chose ten applications from the mibench benchmark suite [40]. We compared the running time for the applications using two configurations. The first one is *Base* which acts as the control group where the execution of the original code runs on the host processor with the ROP monitor disabled, thus being exposed to ROP attacks. *Ours* refers to the same code execution with the monitor enabled. We

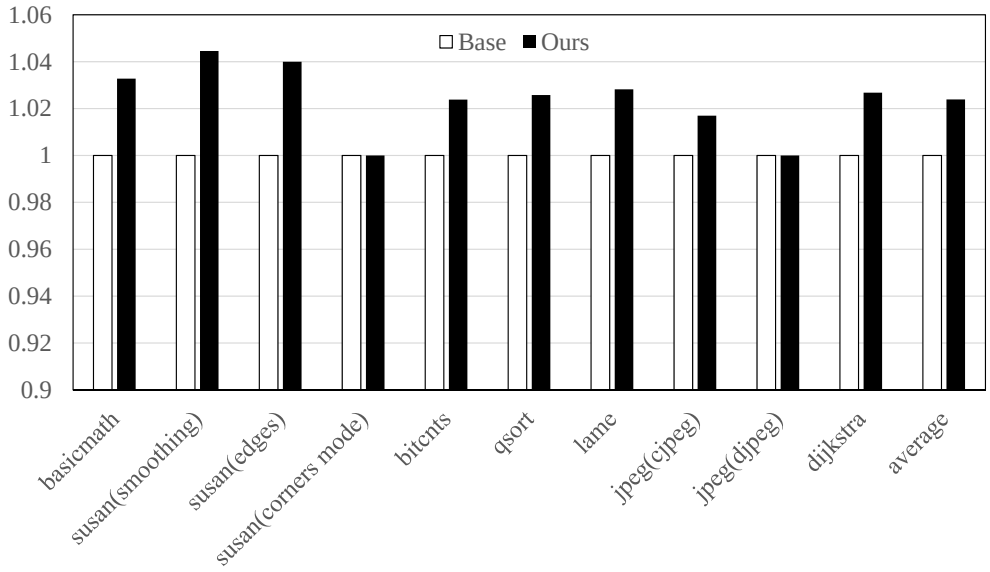


Figure 3.7 Comparison of the execution time normalized to the Base configuration

show the performance numbers of the two configurations in Figure 3.7 where the execution time of each configuration is normalized to that of *Base*. The results evince that our ROP monitor only incurs 2.39% running time overhead on average over *Base*. This performance overhead is caused by resource conflicts between the host CPU and our monitor because they share the same memory as explained in Section 3.

To evaluate the detection capability of our ROP monitor, we have implemented three types of ROP attacks based on the Shell-storm shellcode [85], as shown in Table 3.2. A1 and A2 are crafted to open a new shell, thus enabling attackers to enter arbitrary commands. A3 changes the attribute of the memory page where the attacker's own code is located with the `mprotect` system call. Among them, A2 contains a long-gadget which enables the attack to bypass the signature-based CRA defense mechanisms [21, 46, 68], which use short gad-

get lengths as a distinctive feature of CRAs. To gather necessary gadgets, we have leveraged three general libraries (libwebcore in Android 4.2.2, libc-2.13 in Xilinx-linux and libc-2.15 in Ubuntu) as our code base.

With the implemented attacks, we have tested the effectiveness of our monitor in terms of security. As we expected, all the implemented ROP instances are detected by SCS. Since ROP attacks violate the general convention of the function invocation, their malicious behaviors are always observed by our detection scheme even when the advanced skills like long-gadgets are employed. Based on this result, we assert that our ROP monitor can protect the target system from any type of ROP attacks.

Attack No.	Goal	Advanced Skill	Detection
A1	Open a shell	-	√
A2	Open a shell	Long-gadget	√
A3	Invoke a mprotect system call	-	√

Table 3.2 Description of implemented ROP attacks and detection results of the attacks

3.6 Chapter Summary

This chapter introduces a hardware-based ROP monitor to detect code reuse attacks on commodity smart mobile devices. The monitor provides a negligible performance overhead for runtime detection of ROP attacks. Moreover, the monitor has been implemented without any modifications in the processor internal, and the hardware modules are integrated with a widely available processor core, observing the conventional AP design rules so that our solution can be easily implanted to commercial mobile APs. Our experiments on the FPGA prototype revealed that our current implementation successfully de-

tects synthetic ROP attacks. The experimental results also reveal that the area overhead of the hardware for our monitor is acceptably small even when being compared to the normal sizes of today's mobile processors. All in all, we hope that our proposed architecture would become an attractive CRA defense solution to production-quality ARM-based mobile AP platforms.

Chapter 4

Efficient Security Monitoring with Core Debug Interface in an Embedded Processor

4.1 Introduction

In this chapter, we propose a systematic way to integrate the external hardware monitor engines with the host processor, which can resolve the problems of my previous studies. Our approach is similar to those in [19, 44, 59] in that the monitor engines are also connected externally to the host processor. But looking at the details, ours is different from them in several aspects, about which more discussion will be given in Section 7. One striking difference is that our approach does not modify internally the host processor architecture to install a customized interface or connection for the external monitor engines. Instead, in our system, the engines are connected to the host processor via an existing standard interface, called the *core debug interface* (CDI), which has been readily available for debugging purposes in various modern processors including the

commodity ones like ARM Cortex series and Intel x86 architectures with Processor Trace technology [6, 42, 98]. Being plugged into CDI, our monitor engines have full access to the bountiful information transmitted in the form of signals from CDI, with negligible communication overhead.

In principle, a program has two types of flow when it runs, the data flow and the control flow. Since they are closely related to the execution behavior of the program, the monitoring solutions are also largely divided into two subcategories; (1) the data flow monitoring and (2) the control flow monitoring, each of which corresponds to each type of flow. For this reason, to validate the effectiveness of our approach in security monitoring, we chose two representative monitoring methods in our work for each sub-category: (1) DIFT [62] and (2) CFI checking with branch regulation [45]. To realize these schemes on our framework, we design the external hardware engines specialized for the methods and connect them to the host via CDI, thereby enabling them to efficiently perform the security monitoring computation.

To evaluate our approach, we implement and prototype our proposed framework as a full-system on a FPGA board, which incudes the host processor with CDI, the monitor engines for the two schemes and the other supporting modules. In the experiments on the prototype, it is shown that our engine successfully operates at extremely high speed to provide ample protection against various attacks.

In specific, we make the following contributions:

- We propose a framework in which an external monitor engine can efficiently perform the required security monitoring with neither internal host hardware modification nor excessive performance overhead.

- We design the external monitor engines that can cooperate with CDI,

which are specialized for DIFT and branch regulation proposed in [45, 62]. Also, we propose the CDI filter that cooperate with the engines, whose mission is to filter and refine the raw CDI signals.

- We implement a real full-system prototype on FPGA for our approach and evaluate its effectiveness in performance/area cost/security.

This chapter is organized as follows. In Section 2, as the background for our work, we explain the details of CDI, after introducing the principle of the security monitoring techniques implemented in this work. Section 3 gives an overview of our approach. Then, through Section 4 and 5, we describe how we realize the security monitoring techniques on our framework. Section 6 reports our experimental results and in Section 7, we relate our work with others. Finally, in Section 8, we summarize this chapter.

4.2 Background

In this section, for better understanding of our work, we will explain how the well-known security monitoring scheme (the branch regulation [45]) works, which is realized on our framework as we mentioned. (DIFT is already discussed in Section 2.) Then, we explain the details of CDI, which is readily available in many commercial architecture for debugging and can provide the runtime information of a program running on the processor to the security engines.

4.2.1 Control Flow Integrity Checking for Detecting Code Reuse Attacks

To mitigate the threat of CRAs, many solutions have been proposed. Among them, in [1, 45, 100], they have developed the CFI checking techniques that enforce an application to stay in its legal control flow all the times, thereby discouraging attackers' ultimate objective that is, for their profits, to deviate

the execution flow of their victim application from its original and legitimate paths.

Branch regulation introduced in [45] is one of those CFI checking schemes to defend a system from CRAs. In principle, in order to chain the gadgets in CRAs, the attacker makes use of indirect branches. Since there are generally three types of indirect branch instructions in instruction set architectures of processors (i.e., indirect call, indirect jump and return), they propose three types of CFI rules that should comply with at runtime. To prevent the CRAs using return instructions (i.e., return-oriented programming (ROP) attacks), they keep copies of return addresses in a safe space, called as the *shadow stack*, when functions are invoked, and check whether the return addresses have been modified upon function returns. For the protection from the CRAs that uses indirect calls and jumps (i.e., jump-oriented programming (JOP) attacks), they regulate the target of those indirect branches as follows; (a) the target of an indirect call should point to the entry of a function entry and (b) the target of an indirect jump should point to the address within the same function that the jump instruction belongs to. In our work, we also employ the three rules in branch regulation for CFI checking against CRAs, since they are effective in preventing ROP/JOP attacks in the wild [45]. Our implementation based on the rules will be explained in Section 5.

4.2.2 Core Debug Interface

In recent commodity processors, for the efficient debug/trace in real-time without affecting the performance of the target processor, the specialized module for the purpose is supported, so called the on-chip debug (OCD) unit [66]. Provided by OCD, a rich set of information allows developers (or users), on their debugging environment (usually on desktop machines), to follow the path that

the target CPU takes as a result of code execution and monitor values in various registers and memories. Representative examples of OCD are the ARM CoresSight modules [6] supported in ARM Cortex series processors such as the *embedded trace macrocell* (ETM) and the *program trace macrocell* (PTM).

CDI is an interface placed on the CPU side, whose main role is to provide OCD with the CPU's internal status information that is essential for debug/trace. In general, the OCD modules provide various signals to developers such as instruction address, current context ID (or process ID), and data address/value of memory access instructions, which are useful information to keep track of the behavior of a monitored program. Thus, CDI for the OCD modules also provide such information through the dedicated signal lines [6]. As an example, in Figure 4.1, the signals to ETM provided by ARM processors through CDI is described. Although the types of information supported by CDI can vary from one processor architecture to another, the signals presented in Figure 4.1 are generally provided in most CDIs. In Section 3-5, we will discuss how these CDI signals can be utilized for security monitoring schemes in our work.

Signal	**Description**
ETMICTL [20:0]	ETM instruction control bus
ETMIA [31:1]	ETM instruction address
ETMDCTL [10:0]	ETM data control bus
ETMDA [31:0]	ETM data address
ETMDD [63:0]	ETM data write data value
ETMCID [31:0]	Current processor Context ID

Figure 4.1 Description of CDI signals for ETM

4.3 Our Framework

In this section, we will describe the design of our framework, which aims at the efficient security monitoring with the external hardware monitor engines and

CDI. After introducing the overall architecture, we will explain the hardware components that are designed to help the efficient security monitoring in our approach.

4.3.1 Overall Architecture

Figure 4.2 presents the overall SoC design for our framework, which mainly consists of the host processor and the monitoring subsystem. Within our SoC platform, the monitoring subsystem, which includes several components for security monitoring, is basically connected to the host through a generic system bus along with other hardware modules. It has both the master and slave interfaces so that it cannot only respond to the interconnect transactions from other modules, but also initiate transactions to access data in the modules.

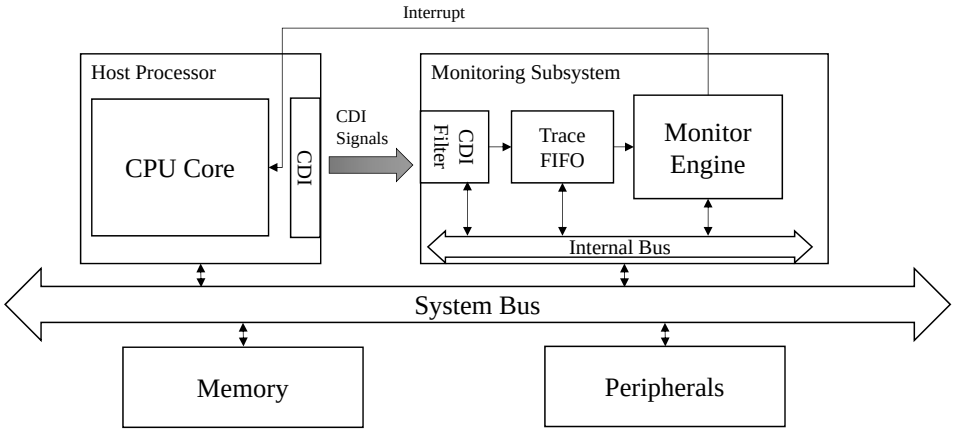


Figure 4.2 Overall SoC platform

In our work, the reason why we use CDI is to support the efficient communication between the host and the monitoring subsystem for runtime information, which is essential for security monitoring. An alternative way for this without CDI is to use the system bus to deliver the information. Of course however, it would consume more bus cycles that normal data transactions could otherwise

use. It should also spend extra CPU cycles in executing instructions for the delivery. For this reason, in this work, to reduce these overheads, we have devised CDI to become a special channel for runtime information such that it can be transmitted from the host into the monitoring subsystem as traces, consuming neither CPU nor bus cycles.

Based on the specification of CDI in commercial processors [3, 6, 98], in our prototype, we assume that CDI provides a set of signals as follows; instruction address, current context ID (or process ID), data address/value of memory access instructions, branch type/source address/target address, exception and privilege mode information. CDI extracts the information from the processor's internal pipeline and they are delivered from the host to the monitoring subsystem through the dedicated channel.

As shown in Figure 4.2, the monitoring subsystem mainly consists of four components; the *CDI filter*, the *trace FIFO*, and a monitor engine that is specialized for a specific monitoring scheme. In the following subsections, we will explain the components in detail.

4.3.2 CDI Filter and Trace FIFO

Although CDI provides a plenty of signals containing runtime information, in most cases, they cannot be simply fed into the monitor engine as they are in their original form, since the signals are generated primarily not for a specific monitoring method but for debugging. Thus, they must be refined and filtered to be what are suitable for a specific security monitoring scheme. For this purpose, we propose a hardware component, called the *CDI filter*, which is located between the monitor engines and CDI. During execution, it takes CDI signals as input and filters the signals properly before delivering them to the monitor engine. For example, in cases of the control flow monitoring schemes, the traces

like data addresses/values accessed by load/store instructions are useless since they do not give any meaningful information about control flow. Therefore, these unnecessary signals are eliminated and the only essential ones are transferred to the trace FIFO, where the runtime traces for the monitor engine are stored. In Section 4 and 5, we will discuss how the CDI filter can be used for a specific monitoring scheme.

4.3.3 Monitor Engine

The monitor engine is the key component of the monitoring subsystem in our framework, which performs the core task of security monitoring. For each monitoring scheme, the CDI filter is configured to filter out unnecessary CDI signals, thereby leaving only the essential information to the trace FIFO. Then, the security engine carries out its task with the traces in the FIFO, following the execution of the host program. In our prototype, when the engine finds any suspicious behavior on the host program, it sends an interrupt to the host in order to report the existence of attacks. Once the OS kernel receives the interrupt, it activates the handler routine to take action against the attacks.

In our proposed framework, the security engine is a hardware component specialized for a specific monitoring scheme. In the following sections (Section 4 and 5), we will discuss how the monitoring engine can be designed to operate on our framework, while coping with CDI of the host processor.

4.4 Bulding a DIFT Engine for CDI

In this section, we will first describe how DIFT for DLP is implemented on our framework. Then, we will explain the detailed structure of our DIFT engine, which is designed to efficiently perform the DIFT computations while coping with CDI.

4.4.1 DIFT on Our Framework

In section 2, we introduced the three steps of DIFT; tag initialization, tag checking and tag propagation. To perform DIFT for DLP, in our system, the host OS kernel takes responsibility of the first two stages (tag initialization and checking), and our engine of the last one partially because tag propagation is the pivotal and most time-consuming task in DIFT. In our work, we associate a tag bit to each data location such as registers and memory locations, like other DIFT approaches [62, 73, 92]. The all tags are basically managed by the DIFT engine and the host should access them through the device driver for the engine.

In this work, we assume the attackers who attempt to access the files that contains sensitive information like password. Thus, in our system, these files are labeled as the sensitive sources, in order to be monitored with DIFT. For tag initialization, we have modified system calls for file accesses, such as **open**, so that the kernel can be aware of every access of an application to any file in the system. If any file is accessed at runtime, the kernel compares the file name to the *list of sensitive files*. When it is in the list, the tag for the corresponding file pointer is tainted. Since this step requires interaction between the host and the DIFT engine, we have implemented a tag initialization function in the device driver that basically reports to the engine the location (i.e., register number or memory address) of the data that need to be tainted. Then the engine, in return, taints the associated tags for the location in the report delivered from the kernel.

For tag propagation, any data derived from the file is tainted to denote being sensitive because its file pointer tag is on. A tag is propagated in a machine instruction from a source operand to the destination operand based on a set of

tag propagation rules which are specified at the granularity of basic operations such as arithmetic and logical operations. Figure 4.3 shows a segment of the host code and its associated propagation rules with operands. In the figure, the propagation rule at line 2 in (b) depicts that the **or** operation needs to be performed on the tags of **r9** and **r3** before the result is propagated to the tag of **r3**. In short, two propagation rules, **or** and =, are applied when the original code at line 2 is executed. From this example, we learn that for the generation of a tag propagation rule, the DIFT engine must decode the given instruction and identify its opcode and operands.

The tag propagation task on our engine is basically determining whether each tag should be on or off as the host code executes. Every time the host executes an instruction, the engine also carries out the corresponding propagation rule like those shown in Figure 4.3. Since this rule is generated from each instruction at runtime, the engine first fetches the same instruction from the main memory that the host CPU just did, and tries to resolve the operand values in order to locate every tag operand for its tag operations. However, not all values can be resolved only by decoding the instruction. For instance in Figure 4.3, the load instruction at line 1 uses two operands: register and memory. For correct tag operations, the engine must have the exact register number and memory address. While the former is trivially found (i.e., **r9**) right from the instruction, the latter remains unknown since the value of **r0** is hidden inside the host CPU. In addition, only with the program code, the execution path at runtime cannot be obtained although it is necessary to fetch the same instructions the host processor executes. In our system, therefore, such hidden information is forced to flow from the host into the DIFT engine as runtime traces. In our implementation, we configure the CDI filter not to filter out the related signals (i.e., the memory addresses accessed by load/store instructions, branch target

	Original Code	Tag Propagation
1	ldr r9, [r0, #0x40]	tag[r9] = tag[r0] or tag[deref[r0]]
2	add r3, r9, r3	tag[r3] = tag[r9] or tag[r3]
3	orr r9, r9, #0xc0	tag[r9] = tag[r9]
4	sub r2, r9, #0xf	tag[r2] = tag[r9]
5	str r2, [r1]	tag[deref[r1]] = tag[r2]
	(a)	(b)

Figure 4.3 Example of tag propagation rules

addresses) so that the DIFT engine can acquire the essential information from CDI signals.

Sensitive information can be leaked through various channels, such as network connections or USB ports. Thus, for tag checking, in our system, we installed a function into the system calls involved in data output channels, such as network packet generation. When data is to be carried outside in a network packet through an output channel, this kernel function checks the data tag with the assistance of our DIFT engine. As the first step of this check, the function makes an inquiry to the engine with the location of the data being transferred out. Upon receiving the inquiry, the engine checks the tag value, and notifies the host of the tag checking result. Once the kernel receives the result, it finally checks whether the data is leaked as part of legitimate operations or not. If the tainted data is leaked as a result of unauthorized operations, the kernel raises an alarm. Note that deciding the legitimacy of certain operations is in fact beyond the scope of this chapter as it is irrelevant to the design of our DIFT engine.

4.4.2 Design of our DIFT Engine

Figure 4.4 presents the internal block diagram of our DIFT engine, which manages all tags and perform the task of tag propagation. The *main controller*

governs the communication between the host and the engine as well as all transactions related to DIFT computation. It is configured by the host to control the DIFT engine, through the device driver. By setting the values of the controller registers, the host can direct the operations of the engine, such as initialization and assignment of the functions for the *tag propagation unit* (TPU). As the central component of our engine, TPU processes all the tags that are associated with data storage in the host. Each entry in the *tag register file* (TRF) represents the tag for an individual register in the host CPU. Borrowing the idea from [92], the engine reserves a special region, called the *tag space*, in memory to stores a long array of bits each of which represents one word of host data in memory. To reduce the memory latency for accessing the tag space, TPU has a small cache, called the *tag cache* [44], for frequently accessed memory tags.

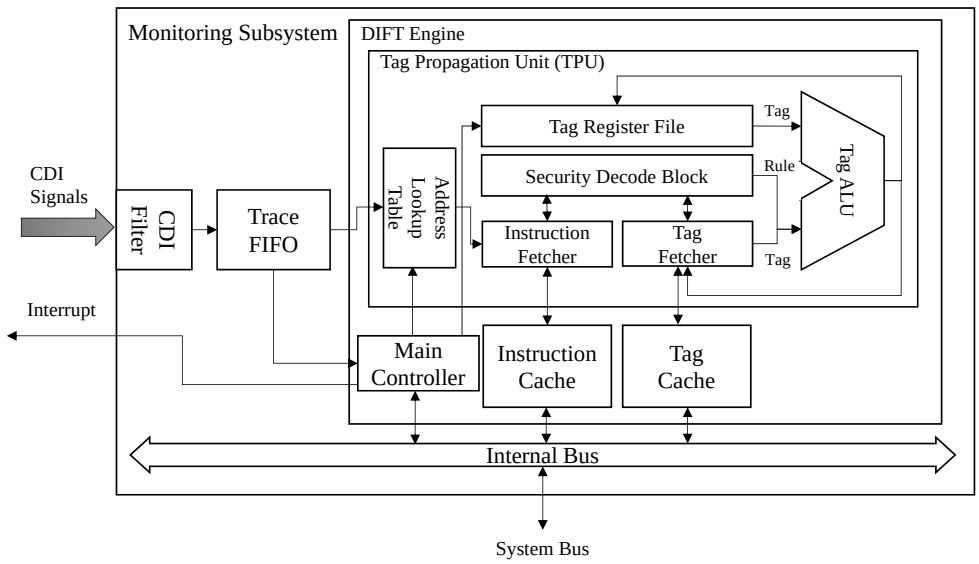


Figure 4.4 Microarchitecture of our DIFT engine

For unerring tag propagation, it is crucial that TPU correctly follows the execution of the host program. Although most necessary information can be ob-

tained from the host code, as we mentioned, the two types of information should be given to our engine at runtime; (1) the execution path and (2) load/store addresses. To complement these information, in our current DIFT implementation, we configure the CDI filter to leave the following signals and eliminate the others; the current process ID (PID), the address of memory data accessed by a load/store and the target address of a branch. (The current PID is necessary to recognize the active process running on the host. If the monitored program becomes in the sleep mode, the main controller has the trace FIFO to ignore the traces from the CDI filter.) At runtime, these traces are stored to the trace FIFO in order, then our engine consumes them to obtain necessary information.

With the branch target addresses from CDI, the engine can be aware of the control flow of the host. Based on this, it reads the instructions at the address from the main memory, which were just executed by the host. Then, every fetched instruction enters the *security decode block* (SDB), which derives a tag propagation rule from the opcode and operands of the instruction. If the operand is a register, TPU reads from TRF the tag register value corresponding to the operand. If the operand is the memory address for a load/store with register-indirect addressing (see Figure 4.3), TPU acquires the exact address by reading a trace from the trace FIFO. (Since all load/store instructions generate the traces for the access addresses and they are stored in the trace FIFO in order, it is guaranteed that TPU can obtain the address for the memory instruction.) Then, it loads from the tag cache the tag bit representing the memory address. If a tag cache miss is taken place, the *tag fetcher* accesses the tag space allocated in the main memory to fill the requested line. Once all the tags are ready, TPU performs the tag propagation for the fetched instruction, and writes the result back to the tag bit for the destination operand in the instruction. If the fetched instruction is a branch, TPU takes a trace from the

trace FIFO to obtain the next branch target address and reads the instructions at the address.

The idea of making TPU follow execution trails of the host brings about a couple of design challenges. One of them is that to follow the trails, TPU relies on the address values carried in the runtime traces, but the values are virtual addresses while TPU uses physical addresses to access the host memory. To resolve the discrepancy in these address spaces, we have the *address lookup table* (ALT) in TPU. An entry of ALT consists of the PID for a process running on the host and the virtual-to-physical address mapping information for the corresponding process. The mapping is determined by the host OS kernel when a new page is allocated for the code section of a process. Therefore, we have slightly modified several system calls related to page allocation in a way that whenever a page is allocated for a process, the mapping information along with its associated PID can be forwarded to TPU for ALT update. Fortunately for our design, a process usually holds only a few entries in ALT. This is because the code section ordinarily occupies a smaller number of pages than the data section. When a process is terminated, its entries are removed from ALT. For this procedure, we have also altered relevant system calls like **exit()**.

Another challenge here is that if TPU should always fetch instructions from memory, it could not catch up with the CPU speed certainly because memory is slow. To tackle this, we have the instruction cache (I-cache) in the DIFT engine. When TPU fetches an instruction, it first tries to load it from I-cache. If a miss occurs, TPU commands the *instruction fetcher* to read the entire cache line containing the instruction from the main memory. By employing I-cache, the memory latency required for fetching instructions can be substantially reduced.

4.5 Implementing a CRA Detection with CDI

In this section, we will explain our implementation of branch regulation, which is a well-known solution against CRAs (especially for ROP/JOP attacks). Then, we will present the detailed structure of our CRA detection engine that efficiently performs the control flow monitoring on our framework.

4.5.1 Branch Regulation on Our Framework

The objective of branch regulation is to prevent the two types of CRAs, ROP and JOP attacks [45]. To achieve its goal, as we introduced in Section 2, branch regulation employs the two types of defense mechanisms, each of which is for a CRA type; the shadow stack to check the legality of return instructions and the regulation scheme for indirect calls/jumps. In our implementation, we also build the two mechanisms to employ branch regulation.

To detect the malicious branch behaviors of ROP attacks with the shadow stack, the two primitive operations should be performed as follows:

1. When a function is invoked, the corresponding return address is duplicated and pushed into both the real program stack and the shadow stack.

2. When a function returns to its caller, the most recent return address is popped from the shadow stack, and compared to the actual return address saved in the program stack. If there is a mismatch between the two addresses, it is ruled that the actual address is maliciously manipulated by the attacker. With the rule checked on every function return, it was proven that any illegitimate use of a return, especially due to the an attack empowered by ROP, can be detected [26].

In our implementation, we install the shadow stack in our CRA detection engine to perform these two operations. At runtime, according to the control

flow information flowing from CDI, the shadow stack is managed and the engine checks the validity of return addresses. In order to carry out the operation (1), our engine should be aware of the occurrences of function calls and the addresses at which the function is located. Since the CDI of our prototype provides the type and the source address of every branch instruction (as explained in Section 3), our engine can acquire the required information. When a branch instruction is executed on the host and it is a function call, the engine takes the source address of instruction. Then, it pushes the return address of the function call (i.e., (the source address + 4) in 32-bit instruction-set) to the shadow stack. For the operation (2), our engine should be aware of the occurrences of returns and the associated target addresses. Similarly as in the case of the first operation, the necessary information can also be obtained from the CDI signals. When a return is executed on the host, the engine will receive the necessary information for the operation (2), then check the validity of the return addresses, comparing it with the top entry of the shadow stack.

Our CRA detection engine also performs the second defense mechanism of branch regulation, whose purpose is to check the legality of indirect calls and jumps, based on the simple invariants ruling the normal behaviors of branches in a programming language. As explained in Section 2, for the indirect calls/jumps, the target address is restricted to a point in the current function or the entry of a function. Thus, in order to perform this scheme, our engine must know both the beginning and ending addresses of a function body. For this, like other studies [24, 97, 100], we conduct off-line binary analysis for a monitored program to generate and store the boundary information in the form of tables. When an indirect call or jump is executed on the host, the engine accesses the summarized information from the main memory and uses it to check the validity of the target address, which is acquired from CDI signals.

4.5.2 Design of our CRA Detection Engine

In this subsection, we discuss the detailed architecture of our CRA detection engine, which performs the two defense mechanisms introduced in the previous subsection. In Figure 4.5, the structure of our monitor engine for CRA detection is depicted. Our CRA detection engine consists of three main components; (1) the *main controller*, (2) the *secure call stack* (SCS) and (3) the *indirect branch bounds checker* (IBBC).

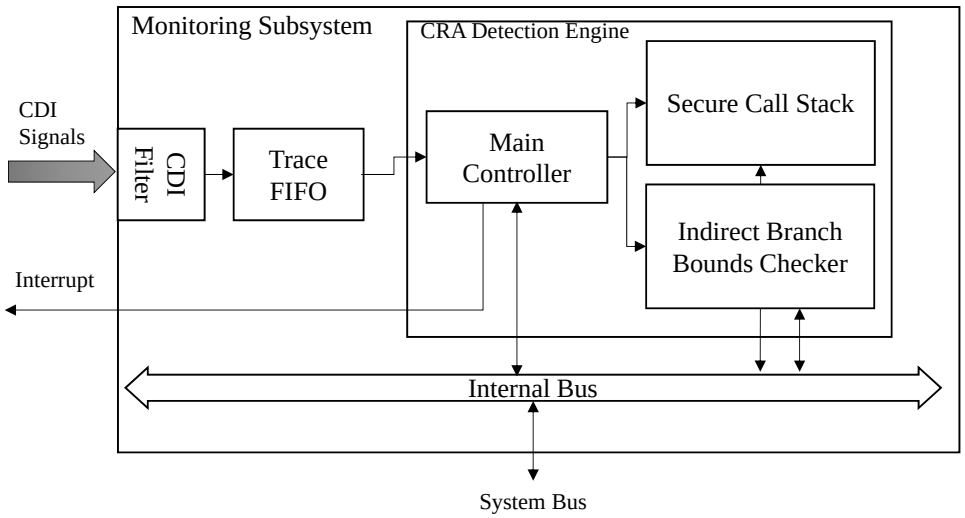


Figure 4.5 Microarchitecture of our CRA detection engine

For branch regulation, it is unnecessary to monitor all branches executed on the host. Rather, according to the defense mechanisms, only the three types of branch should be traced (function calls, returns and indirect jumps). Thus, in our implementation, the CDI filter leaves only the traces associated to the three types of branches in the trace FIFO (i.e., the type, source address, target address for the three types of branches). Once traces for a branch is delivered to our engine, the main controller firstly checks its type and according to the type,

it controls the engine to perform the defense mechanisms. When a violation on the branch behavior is detected by SCS or IBBC, which actually carry out the defense mechanisms, the main controller generates an interrupt signal to inform the host processor of it.

Shadow Call Stack

SCS in our engine conducts the task of investigating the validity of return target addresses to detect ROP attacks. Figure 4.6 shows the hardware architecture of SCS with three input signals flowing from the main controller; one is `addr_in` for the return address of a branch instruction, and the other two are `call` and `return` for branch types. Using the latter two inputs, the *queue controller* composes the three signals `push`, `pop` and `counter`, and forwards them to the *address queue*, whose job is to maintain the shadow stack. When a function is called, the controller sets `push` on and stores the value `addr_in+4` into its queue. At the next cycle, it increases the counter value by one. When the function returns, the controller sets `pop` on, outputs the front queue value through the `addr_out_queue` line, and decreases the counter value by one.

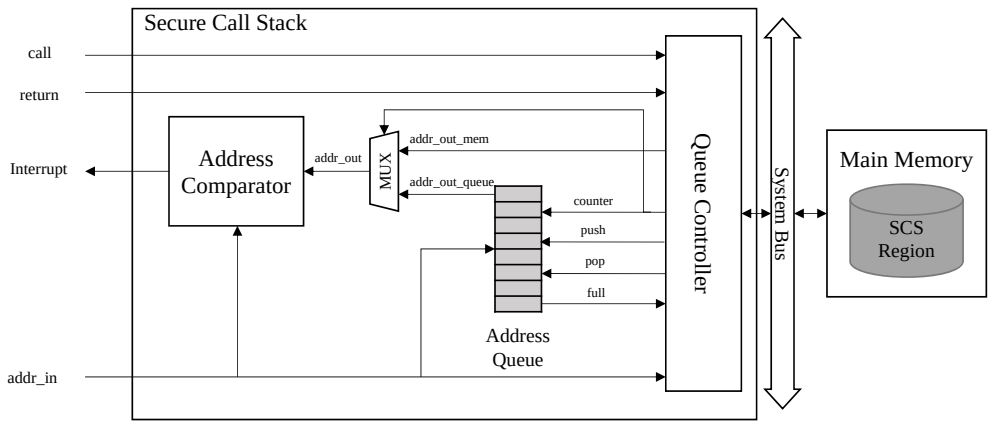


Figure 4.6 SCS architecture

Note that the address queue has a finite number of entries (16 in our current prototype). Due to this limited number of its entries, the queue will be overflown if the host code contains more than sixteen times nested calls. To cope with this exceptional case, we reserve a memory region, called the *SCS region*, in the main memory. When the queue is full, its controller saves into the SCS region the return addresses for all subsequent function calls. Later when one of these functions returns to its caller, the controller fetches the saved return address from the SCS region and outputs it via the `addr_out_mem` line.

Figure 4.6 shows a multiplexer that is connected to two input lines, `addr_out_queue` and `addr_out_mem`, respectively from the different sources for saved return addresses. The multiplexer chooses from which source a return address is transferred to the output line `addr_out`. The choice depends on the value of `counter`. For the value ≤ 16, the queue should be the source. Otherwise, the other source will be chosen. The destination of `addr_out` is the *address comparator* that is in charge of making a final decision about the existence of an ROP attack. For this, the comparator takes two input values from `addr_in` and `addr_out`. The former value represents the actual address that the current function is about to return, and the latter the original return address saved when the function was called. If these values do not match, the comparator interrupts the main controller to notify of an ROP attack. To summarize, in our design, the value of `addr_in` is pushed into either the address queue or SCS region at a call instruction, and compared at a return instruction with the value of `addr_out` from the address queue.

Indirect Branch Bounds Checker

The role of IBBC in our CRA detection engine is to check the legitimacy of function calls and indirect jumps, based on the rules of branch regulation. As

shown in Figure 4.7, IBBC receives from the main controller the three signals, `call`, `indirect_jump` and `addr_in`. The first two signals are transmitted when a call or an indirect jump is executed on the host. The last is to carry to IBBC the target address of a branch associated with one of these two signals arriving simultaneously. As we stated earlier, we perform the offline binary analysis of a target application a priori to create the meta-data set that will inform IBBC of all the function boundaries in the application code. For each instruction in the host code, the two types of meta-data are generated; (1) a bit that indicates whether the location is an entry of a function or not, and (2) the size of the function that the instruction belongs to. When the application begins to run, the data set is placed into the location pre-allocated in the main memory.

At runtime, the *meta-data manager* waits for the `addr_in` signal from the main controller. When the signal finally arrives, the manager first extracts the target address and reads the meta-data elements which corresponds to the branch instruction, in order to create the two forms of output signals, `func_entry` and `func_size`. These signals together with those directly from the main controller are given as input to the *function bounds checker* (FBC) whose mission is to catch any branch instruction in a function jumping across the function's boundaries. Upon receiving `call`, FBC becomes aware that there has just been a function call made in the host. Then, it will immediately read `addr_in` to get the target address of the call instruction, and check if the address points to a known function entry by reading `func_entry` for the address. The signal value should be either non-zero for a function entry or zero for other addresses (i.e., meta-data (1)). Therefore, if it is found to be non-zero, FBC will understand that the current call targets a legitimate function entry, and so calculate the end address of the callee function by adding the value of `func_size` to its entry address `addr_in`. On the other hand, if `func_entry` = 0, it means

that the call is about to jump to an unknown address, which is a typical behavior exhibited by a JOP attack. Thus in this case, FBC interrupts the main controller to report the attack. Although the meta-data is not frequently accessed, due to the latency of the main memory, it can degrade the performance of our engine. In order to mitigate the overhead, we install the meta-data cache, which is located between the meta-data manage and the main memory.

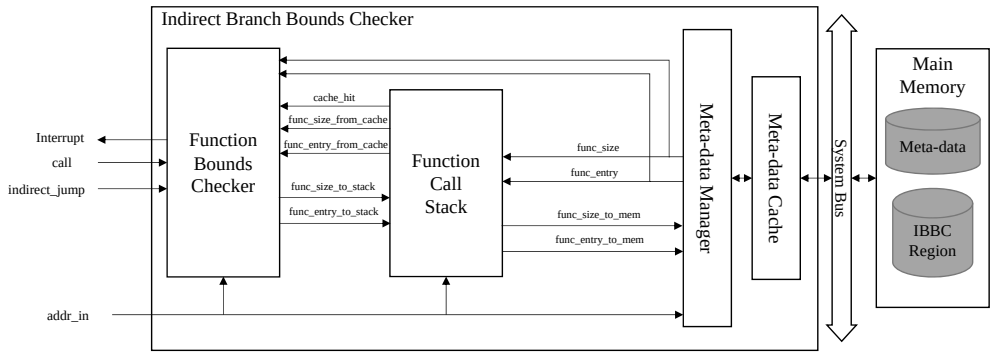


Figure 4.7 Block diagram of IBBC

For the case where function calls are nested, we provide a separate storage, called the *function call stack* (FCS), to save the calculated entry and end addresses of the earlier callees. Whenever a function calls another function, the calculated function bounds for the caller is stored in FCS. When a function returns to its caller, FBC pops the top entry of FCS that contains the function bounds for the caller, and uses it for the new function bounds. When FCS is full and a function is invoked, the meta-data manager saves the entry and end addresses of the callee function in the pre-defined region, called the *IBBC region*, in the main memory. In this case, when the function returns later, the manager reads the saved function bounds from IBBC region and send them via the `func_size` and `func_entry` lines.

When an indirect jump is made in the host, the main controller posts the

event by sending `indirect_jump` to FBC. Then FBC will check if the target address `addr_in` falls between the entry and end addresses of the current function which have already been calculated when the function was invoked initially. If the address points to outside the function boundaries, FBC deems that this is the act of a JOP attack, and spontaneously alerts the main controller.

4.6 Experiment

In this section, we will present the experimental results evaluated on our prototype, which contains the two security engines (i.e., the DIFT engine and the CRA detection engine) as well as the host system. After reporting the synthesis results of our hardware prototype, we will show the security and performance evaluations for each security monitoring engine.

4.6.1 Prototype and Synthesis Result

To evaluate our approach, we have built a full-system FPGA prototype, where the host processor is the SPARC V8 processor, a 32-bit synthesizable core [55] which uses a single-issue, in-order, 7-stage pipeline. It has separate 4K-byte 2-way set associative instruction and data caches. Even though our host processor core provides its own CDI specification [55], the information that comes out of the CDI is quite limited when it is compared to that of commercial products, such as ARM. Thus, we slightly augmented our core to support the CDI signals, as described in Section 3, which resemble those for the ETM of ARM [3]. The bus compliant with AMBA2 AHB protocol [51] is used to interconnect all the modules in our prototype system. Linux 2.6.21.1 is used as our OS kernel and it has been slightly modified to provide supports for our security engines (e.g., the device driver to control the engines).

The security monitor engines (each for DIFT or branch regulation) are im-

plemented as described in Section 4 and 5. In our DIFT engine, we install the tag cache which is a 512-byte, 2-way set-associative cache with 4-byte cache lines, and the DIFT instruction cache which is a 4K-byte, 2-way set-associative cache with 32-byte cache lines. The CRA detection engine also contains the meta-data cache which is a 512-byte, 2-way set-associative cache with 4-byte cache lines.

Based on the parameters for the prototype as described above, we synthesized our overall SoC Design onto the prototyping board with a Xilinx XC5VLX330 FPGA and 64MB external SDRAM. Table 4.1 provides the design statistics of our hardware prototype. We quantified the resources necessary for the two security engines implemented on our framework, in terms of lookup tables for logic (LUTs) and block RAMs. The design statistics shows that, compared to the baseline SPARC core, the DIFT engine incurs the resource overhead of 60.0% and 27.98% for BRAMs and LUTs, respectively. The CRA detection engine also requires a similar level of overhead (15.00% for BRAMs and22.34% for LUTs). These seemingly large overheads are attributed mainly to the small size of our tiny host core, employed for our academic research, which only consumes approximately 25,000 gates [55]. Considering that recent SoC platforms deploy more complex processors like ARM Cortex series, we assure that the area overhead due to our security engines is acceptable in a more realistic machine.

4.6.2 Experimental Results for DIFT

Security Evaluation

To test the security capability of our DIFT engine, we have synthesized the malware that encrypts a sensitive file named as "secret.txt" and passes it through the network. Our malware, which is similar to the *Dorifel* [47] malware in the

Category	Component	LUTs	BRAMs	DSP48E
Baseline System	SPARC V8 Core (Host Processor)	4876	18	4
	Bus components (AHB Buses + AHB/APB bridges)	439	0	0
	Memory Controller	405	0	0
	Peripherals (TIMER, UART, Interrupt Controller and etc.)	963	2	1
	Total Baseline System	**6683**	**20**	**5**
DIFT Engine	Address Lookup Table	670	0	0
	AHB Master IF	154	0	0
	CDI Filter	27	0	0
	FIFO	129	0	0
	Instruction Cache	293	10	0
	Instruction/Tag Fetcher	97	0	0
	Main Controller	176	0	0
	Security Decode Block (SDB)	35	0	0
	Tag ALU	109	0	0
	Tag Cache	180	2	0
	Total DIFT Engine	**1870**	**12**	**0**
	% DIFT Engine over Baseline System	**27.98%**	**60.00%**	**0.00%**
CRA Detection Engine	CDI Filter	48	0	0
	Trace FIFO	132	0	0
	Main Controller	538	0	0
	Secure Call Stack	46	1	0
	Indirect Branch Bounds Checker	574	2	0
	Internal Bus	155	0	0
	Total CRA Detection Engine	**1493**	**3**	**0**
	% CRA Detection Engine over Baseline System	**22.34%**	**15.00%**	**0.00%**

Table 4.1 Synthesis result

wild, has the ability of evading an intrusion detection system or signature-based DLP solution by using the AES encryption algorithm. However, as planned, any attempt to access the file from the malware will be detected by our modified **open** system call in the Linux kernel. When being detected, the kernel invoke the tag initialization function to taint the tag of the file pointer and to configure TPU to be ready for tag propagation. The malware naively proceeds and encrypts the data without knowing the existence of TPU, while our DIFT engine keeps track of the information flow by propagating tags. When the malware tries to leak the derived data, it invokes the **send** system call to transmit a

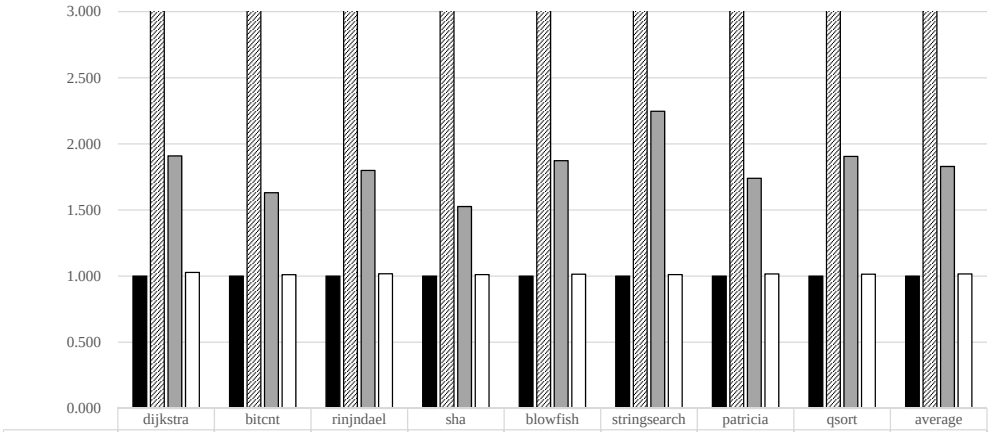
message through the network. Because the system call is also modified to call the tag checking function, the kernel receives tags of the data from TPU as explained in Section 3, and decides whether to allow transfer the data outside or not.

Performance

In order to measure the performance of our DIFT implementation, we chose eight applications from the *mibench* benchmark suite [40]. The performance of our DIFT engine with CDI is compared with those of three systems that have different configurations. The first one, called *Native*, stands for a system that executes the original code with DIFT disabled. The *Software-only* solution employs a software-instrumentation technique to augment the host code with instructions that perform DIFT computation on the host. *CDI-DIFT* refers to our DIFT solution that has an external DIFT engine connected to the host side CDI. In addition to these three systems, we added another configuration, named as *Software-DIFT*, that makes use of an external DIFT engine for time consuming tag propagation. The only difference compared to our solution is that the external DIFT engine does not have a connection via CDI to the host. Therefore, for the engine, the host must execute additional instructions to explicitly transmit runtime traces through the system bus. For this, we instrumented the host code with a set of instructions each of which is inserted after every branch and load/store instruction to send the updated traces to the DIFT engine. We used our in-house tool for code instrumentation.

In Table 4.8, we present the performance comparison of the four configurations. In the table, the host execution time of each configuration is normalized to that of *Native*. The results show that the *Software-only* solution suffers from an excessive performance overhead in that the total runtime is on average 23

times slower than that of *Native*. The overhead of *Software-DIFT* is less devastating than the *Software-only* version: it shows drastically reduced overhead of 82.9% as being compared to that of *Native*. However, it yet runs approximately 1.8 times slower than *Native*. The main cause of such tremendous overhead in both the configurations is the instructions added to the host code for delivering traces. On the other hand, *CDI-DIFT* substantially cuts the overhead down to 1.6% over *Native*. This amazing achievement is mainly due to the fact that, with the supplementary information coming out of CDI, no code instrumentation on the host is needed for our solution. The small amount of performance loss in *CDI-DIFT* is ascribed to the resource competition between the host processor and our DIFT engine because both are connected to the same interconnect and share the main memory.



	dijkstra	bitcnt	rinjndael	sha	blowfish	stringsearch	patricia	qsort	average
Native	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
Software-only	17.785	9.298	47.193	22.556	47.147	17.102	16.290	10.503	23.482
Software-DIFT	1.909	1.631	1.799	1.526	1.873	2.247	1.740	1.905	1.829
CDI-DIFT	1.028	1.011	1.018	1.012	1.015	1.012	1.016	1.015	1.016

Figure 4.8 Comparison of DIFT execution time normalized to Native

4.6.3 Experimental Results for Branch Regulation

Security Evaluation

To evaluate the detection capability of our CRA detection engine, we have implemented four CRA samples (A1-A4), based on [15,16], as shown in Table 4.2. Among them, the two samples (A1, A2) are implemented with ROP technique, and the other two are made as JOP attacks. All of them are crafted to open a new shell, thus enabling attackers to enter arbitrary commands, which is the general purpose of attackers who launches CRAs on the victim system [68].

With the implemented attacks, we have tested the effectiveness of our CRA detection engine as shown in Table 4.2. As we expected, all the implemented ROP samples have been detected by SCS. Since they violate the general convention of the function invocation, their malicious behaviors are always observed by our engine. The implemented JOP samples make use of indirect call or indirect jump instructions to link their gadgets. In these attack samples, every used indirect call instruction does not target an entry of a function. Similarly, all the target addresses of indirect jump instructions are always beyond the current function bounds. Consequently, all their illegal behaviors are detected by our IBBC. Based on this result, we assert that our CRA detection engine provides the same level of security, compared to the original implementation of branch regulation proposed in [45].

Attack No.	Type	Detection	
		SCS	IBBC
A1	ROP	√	
A2	ROP	√	
A3	JOP		√
A4	JOP		√

Table 4.2 CRA Detection Results

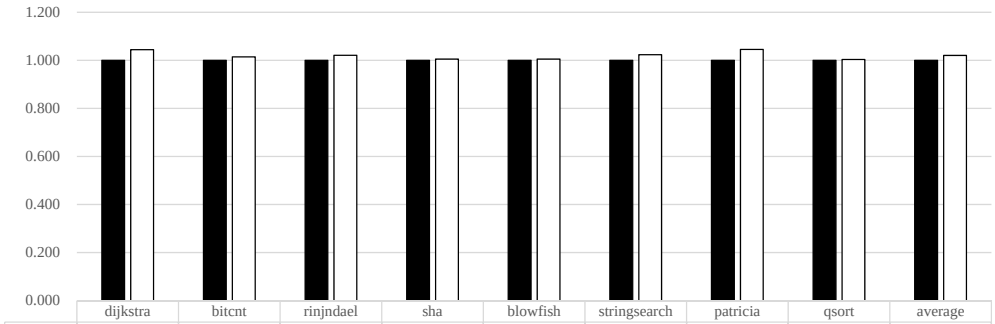
Performance

To measure the performance overhead of our CRA detection engine, we chose the same eight applications of the mibench benchmark suite which are used in our DIFT performance experiment. We compared the running times of the applications using two configurations. The first one is *Native* which acts as the control group where the execution of the original code runs on the host processor with the engine disabled, thus being exposed to CRAs. The other is *CDI-BR* that refers to the same code execution with the CRA detection engine enabled.

We show the performance numbers of *CDI-BR* in Figure 4.9 where the execution time of each application with *CDI-BR* is normalized to that with *Native*. The results evince that our engine with CDI only incurs the overhead about 2% on average over *Native*. This performance overhead is caused by resource conflicts between the host CPU and our CRA detection engine because they share the same main memory, as explained in Section 5. Although our CRA detection engine is equipped with a specialized cache (i.e., the meta-data cache), it occasionally has to access main memory to acquire the meta-data in the main memory, thereby degrading the host performance slightly.

4.7 Related Work

To defend the computer systems from various security attacks, there have been proposed a number of monitoring techniques, such as DIFT [62], memory bound checking [22] and control flow integrity checking [1]. The most popular way to realize security monitoring schemes is to implement them in software. Most software monitoring approaches [1, 20, 29, 62, 73, 79, 82, 99, 101, 104] have relied upon either *source-code instrumentation* or *dynamic binary translation* (DBT)



	dijkstra	bitcnt	rinjndael	sha	blowfish	stringsearch	patricia	qsort	average
■Native	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
□CDI-BR	1.044	1.014	1.021	1.005	1.005	1.023	1.045	1.003	1.020

Figure 4.9 Comparison of CRA detection execution time normalized to Native

[60] for the defense against diverse attacks at execution time. However, the main drawback of them is that they experience excessively high performance overhead. For example, in [62] that proposes a software-based DIFT implementation, the overhead reaches up to about 40 times the original code execution time in the worst case. The performance overhead of DROP [17], which proposes a ROP detection scheme, ranges from 1.9X to 21X. MoCFI [24], which introduces a CFI checking technique on ARM-based mobile devices, shows the performance loss about 5X. Considering that these techniques are usually employed for runtime monitoring, the performance degradation is not acceptable to be deployed in real machines.

To address the shortcoming of software-based monitoring, some early hardware approaches [19,22,23,27–29,43,45,46,56,84,87,92,96,102] tried to improve performance by inserting into the host processor core dedicated hardware modules that accelerate monitoring computations. The main advantage of these approaches is that they do not need to instrument the host code and thus they could bring the overhead down to under 5%. However, they have a disadvantage in that invasive modifications to the processor internal (e.g., registers and

pipeline data paths) are required. For instance in [87], inside the core, they installed hardware tagging units to conduct DIFT, called the *flow tracker* and *tag checker*, and widened the widths of registers, internal datapaths and caches, to accommodate tag bits, all of which call for major changes of the processor internal. In fact, modern microprocessor development may take several years and hundreds of engineers from an initial design to production [27, 44]. Therefore, the substantial costs of development to integrate the customized logic would hamper processor vendors to adopt them, unless the necessity is clearly established.

In an attempt to minimize the internal architecture changes, the researchers in [19, 59] suggested security monitoring solutions in the existing multi-core environment where one general-purpose core is devoted solely to run a *helper thread* that performs tag propagation for the main code running concurrently on a different core. In [41, 44], they proposed an external device that performs monitoring outside the host. By dedicating the monitoring task to a separate core or an external hardware, these approaches can manage to enhance the performance drastically. However, as discussed earlier, the fundamental problem of these approaches is that a vast amount of information must be continuously delivered to the external hardware for accurate monitoring operations [41]. To cope with this communication issue, they modified either the x86 architecture to supplement special hardware queues and new instructions [19, 59], or the CPU pipeline datapath to provide a customized channel between the host and the external device [44]. Our work is somewhat similar to the work in [44] since both propose the external hardware optimized for DIFT. But ours is different from theirs in that we exploit the standard interface CDI for communication. The security engines proposed in our work have been specially designed to perform the monitoring tasks by interpreting the signals for debugging from CDI.

In fact, the idea of using CDI for other purposes has already been explored in several studies, especially in the field of fault-tolerant computing. Some proposed the systems that can inject faults to the host by accessing internal resources such as registers and memory via CDI [33]. Others presented an on-line fault detection technique utilizing CDI to retrieve runtime information in a non-intrusive way [71]. The overall concept of these studies exploiting information flow out of CDI without affecting the state or structure of CPU is similar to ours, but none of the above exploit CDI to perform security monitoring for ensuring system integrity.

To perform CFI checking, several recent works have proposed to leverage the OCD modules that provide branch information [39, 50]. In [50], they propose a ROP detection solution that utilizes the ARM CoreSight Program Trace Macrocell (PTM) to obtain branch information. In [39], the authors introduces another CFI checking solution called CONVERSE, which copes with the IEEE-ISTO Nexus 5001 debugging interface. The difference between ours and these works is that, in our work, we make use of CDI, which provides more various signals than OCD modules. Thus, using the interface, we can implement not only a CFI checking solution (i.e., branch regulation), but also a DIFT one that requires more information beyond branch signals. We also believe that many types of security monitoring can be realized on our framework with CDI, not being limited to the solutions introduced in this chapter.

4.8 Chapter Summary

In this chapter, we presented our novel approach to perform the security monitoring efficiently. Basically in our approach, as in other hardware-based works, the computation-intensive tasks are offloaded to the specialized external hardware engine, thereby reducing the runtime overhead induced by the monitoring

task. However, being located outside the host system, the monitor engine has limited visibility into the host internal states, which becomes a major stumbling block for successful monitoring on the engine. To overcome this limitation, we provide the engine with a separate communication channel through the existing debugging interface, called CDI, of the host. Out of the original CDI signals, only the essential information is filtered so that the engine can conduct its monitoring task efficiently. In addition, it is notable that this efficiency can be achieved without any invasive modification to the host core internal because CDI is readily available in most commercial processor architectures such as ARM.

To show the effectiveness of our approach, in this chapter, we implemented the two well-known security monitoring techniques on our framework;DIFT and branch regulation. Our experiments on the FPGA prototype revealed that the engines successfully detect the attack samples (i.e., data leakage attacks and CRAs). More importantly, our engines attain overwhelmingly low overhead, that is less than 2% for a group of mibench applications. The experiments also revealed that the area overhead of the hardware for our security engines is acceptably small even when being compared to the normal sizes of today's mobile processors. We believe that our approach can be applied to a variety of security monitoring schemes, which have to observe the processor internal states to detect malicious behaviors of attacks.

Chapter 5

Conculsion

This thesis explored the architectural design space for the external hardware engine which aims at the efficient security monitoring. In Chapter 2, I firstly introduced PAU, which supports various monitoring techniques based on the tag processing. With the specialized instructions, PAU can successfully perform the three monitoring methods without any extension in the host processor architecture. However, since the runtime information for monitoring should be delivered via the system bus transactions and the code instrumentation, PAU incurs the non-negligible performance overhead. To resolve the problem, in Chapter 3, I proposed a novel approach that utilizes PTM, the on-chip debug module in the ARM architecture. Being plugged into PTM, the ROP monitor which is the hardware engine for ROP detection can conduct the monitoring task while the performance impact on the host is neglgibile. Nevertheless, this approach requires the meta-data to complement the lack of runtime information from PTM and it consequently incurs the overhead in the memory space and the hardware resources. In the final solution in Chapter 4, I proposed another approach in

order to resolve the problem. The key idea is to connect the hardaware engines to CDI, which is an existing debug interface in modern processor architectures. Since CDI extracts the internal states of the processor from the pipeline and delivers them to the external modules, our external engines can acquire all the necessray information for security monitoring. Thus, the final solution does not suffer from the problem of the previous approach that is caused by the meta-data.

As mentioned earlier, the moremost design priority in this thesis is to derive an external hardware engine architecute that does not require any modification in the host processor while acheiving the efficiency in security monitoring. It is because the previous hardware works that aggressively modify the internal structure of processor are not readily viable in the near future. In this thesis, we explore various design decisions, from the one that only utilizes the system interconnect (i.e. PAU) and another that exploits the exisiting debug technology. I believe that one of the approaches proposed in this thesis can be employed for the security monitoring in commercial products, according to the design rules enforced in the platform.

Bibliography

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.

[2] S. Andersen and V. Abella. Data execution prevention. changes to functionality in microsoft windows xp service pack 2, part 3: Memory protection technologies, 2004.

[3] ARM. *Embedded Trace Macrocell Architecture Specification*, 2011.

[4] L. ARM co. CoreSight Program Flow Trace Architecture Specification, 2011.

[5] L. ARM co. Mali-400 MP, 2012.

[6] L. ARM co. ARM CoreSight Architecture Specification v2.0, 2013.

[7] L. ARM co. ARM System Memory Management Unit Architecture Specification, 2013.

[8] L. ARM co. AMBA Network Interconnect (NIC-301) Technical Reference Manual, 2014.

[9] L. ARM co. Cortex-A9 Processor, 2014.

[10] B. Bailey, G. Martin, M. Grant, and T. Anderson. *Taxonomies for the Development and Verification of digital systems*. Springer, 2005.

[11] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, behavior-based malware clustering. In *NDSS*, volume 9, pages 8–11. Citeseer, 2009.

[12] J. Bellardo and S. Savage. 802.11 denial-of-service attacks: Real vulnerabilities and practical solutions. In *USENIX security*, pages 15–28, 2003.

[13] F. Brasser, B. El Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl. Tytan: Tiny trust anchor for tiny devices. In *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE*, pages 1–6. IEEE, 2015.

[14] D. L. Bruening. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Massachusetts Institute of Technology, 2004.

[15] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: generalizing return-oriented programming to risc. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 27–38. ACM, 2008.

[16] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 559–572. ACM, 2010.

[17] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie. Drop: Detecting return-oriented programming malicious code. In *Information Systems Security*, pages 163–177. Springer, 2009.

[18] S. Chen, B. Falsafi, P. B. Gibbons, M. Kozuch, T. C. Mowry, R. Teodorescu, A. Ailamaki, L. Fix, G. R. Ganger, B. Lin, et al. Log-based architectures for general-purpose monitoring of deployed code. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 63–65. ACM, 2006.

[19] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos. Flexible hardware acceleration for instruction-grain program monitoring. In *ACM SIGARCH Computer Architecture News*, volume 36, pages 377–388. IEEE Computer Society, 2008.

[20] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige. Tainttrace: Efficient flow tracing with dynamic binary rewriting. In *Computers and Communications, 2006. ISCC '06. Proceedings. 11th IEEE Symposium on*, pages 749–754, 2006.

[21] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng. Ropecker: A generic and practical approach for defending against rop attacks. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[22] J. Clause, I. Doudalis, A. Orso, and M. Prvulovic. Effective memory protection using dynamic tainting. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 284–292. ACM, 2007.

[23] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: a flexible information flow architecture for software security. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 482–493. ACM, 2007.

[24] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi. Mocfi: A framework to mitigate control-flow attacks on smartphones. In *NDSS*, 2012.

[25] L. Davi, P. Koeberl, and A.-R. Sadeghi. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference*, pages 1–6. ACM, 2014.

[26] L. Davi, A.-R. Sadeghi, and M. Winandy. Ropdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 40–51. ACM, 2011.

[27] D. Deng and G. Suh. High-performance parallel accelerator for flexible and efficient run-time monitoring. In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, pages 1–12, 2012.

[28] D. Y. Deng, D. Lo, G. Malysa, S. Schneider, and G. E. Suh. Flexible and efficient instruction-grained run-time monitoring using on-chip reconfigurable fabric. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 137–148, Washington, DC, USA, 2010. IEEE Computer Society.

[29] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic. Hardbound: Architectural support for spatial safety of the c programming language. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 103–114, New York, NY, USA, 2008. ACM.

[30] U. Dhawan, C. Hritcu, R. Rubin, N. Vasilakis, S. Chricescu, J. M. Smith, T. F. Knight Jr, B. C. Pierce, and A. DeHon. Architectural support for software-defined metadata processing. 2015.

[31] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[32] Z. B. Features. Zc702 evaluation board features. *ZC702 Evaluation Board for the Zynq-7000 XC7Z020 Extensible Processing Platform*, 2012.

[33] A. V. Fidalgo et al. Real-time fault injection using enhanced on-chip debug infrastructures. *Microprocessors and Microsystems*, 35(4):441–452, 2011.

[34] A. V. Fidalgo, M. G. Gericota, G. R. Alves, and J. M. Ferreira. Real-time fault injection using enhanced on-chip debug infrastructures. *Microprocessors and Microsystems*, 35(4):441–452, 2011.

[35] V. Gaikar et al. IPhone 4S officially announced by Apple. 2011.

[36] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *NDSS*, 2004.

[37] M. Graphics. Modelsim, 2007.

[38] D. C. U. Guide. Version c-2009.06. *Synopsys.(a)(b)(c)*, 2009.

[39] Z. Guo, R. Bhakta, and I. G. Harris. Control-flow checking for intrusion detection via a real-time debug interface. In *Smart Computing Workshops (SMARTCOMP Workshops), 2014 International Conference on*, pages 87–92. IEEE, 2014.

[40] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14. IEEE, 2001.

[41] H. Ingoo, K. Minsu, L. Yongje, C. Changho, L. Jinyong, K. Brent Byunghoon, and P. Yunheung. Implementing an application specific instruction-set processor for system level dynamic program analysis engines. *ACM Trans. Des. Autom. Electron. Syst.*, 2015.

[42] Intel. Intel 64 and IA-32 Architectures Software Developer Manuals, 2015.

[43] J. A. Joao, O. Mutlu, and Y. N. Patt. Flexible reference-counting-based hardware acceleration for garbage collection. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 418–428. ACM, 2009.

[44] H. Kannan, M. Dalton, and C. Kozyrakis. Decoupling dynamic information flow tracking with a dedicated coprocessor. In *Dependable Systems Networks, 2009. DSN '09. IEEE/IFIP International Conference on*, pages 105–114, 2009.

[45] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev. Branch regulation: Low-overhead protection from code reuse attacks. In *Com-*

puter Architecture (ISCA), 2012 39th Annual International Symposium on, pages 94–105. IEEE, 2012.

[46] M. Kayaalp, T. Schmitt, J. Nomani, D. Ponomarev, and N. Abu-Ghazaleh. Scrap: Architecture for signature-based protection from code reuse attacks. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 258–269. IEEE, 2013.

[47] K. Lab. DORIFEL MALWARE ENCRYPTS FILES, STEALS FINANCIAL DATA, MAY BE RELATED TO ZEUS OR CITADEL, 2012.

[48] H. Lee, H. Moon, D. Jang, K. Kim, J. Lee, Y. Paek, and B. B. Kang. Ki-mon: A hardware-assisted event-triggered monitoring platform for mutable kernel object. In *Presented as part of the 22nd USENIX Security Symposium*, pages 511–526, Washington, D.C., 2013. USENIX.

[49] J. Lee, Y. Lee, H. Moon, I. Heo, and Y. Paek. Extrax: Security extension to extract cache resident information for snoop-based external monitors. In *Design, Automation and Test in Europe Conference and Exhibition, 2015. Proceedings*. IEEE, 2015.

[50] Y. Lee, I. Heo, D. Hwang, K. Kim, and Y. Paek. Towards a practical solution to detect code reuse attacks on arm mobile devices. In *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy*, page 3. ACM, 2015.

[51] A. Limited. AMBA Specication, 1999.

[52] A. Limited. Procedure call standard for the arm architecture, 2012.

[53] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program

analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[54] I. Majzik. Concurrent error detection using watchdog processors. In *Fault tolerant computing systems*, volume 283. Kluwer Academic, 1996.

[55] L. P. U. Manual. Gaisler research. 2004.

[56] A. Meixner, M. E. Bauer, and D. J. Sorin. Argus: Low-cost, comprehensive error detection in simple cores. In *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, pages 210–222. IEEE, 2007.

[57] H. Moon, H. Lee, J. Lee, K. Kim, Y. Paek, and B. B. Kang. Vigilare: Toward snoop-based kernel integrity monitor. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 28–37, New York, NY, USA, 2012. ACM.

[58] S. Mysore, B. Mazloom, B. Agrawal, and T. Sherwood. Understanding and visualizing full systems with data flow tomography. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 211–221, New York, NY, USA, 2008. ACM.

[59] V. Nagarajan, H.-S. Kim, Y. Wu, and R. Gupta. Dynamic information flow tracking on multicores. In *Proceedings of the Workshop on Interaction between Compilers and Computer Architectures*, 2008.

[60] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan Notices*, 42(6):89–100, 2007.

[61] T. Newsham. Format string attacks, 2000.

[62] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.

[63] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn. Parallelizing security checks on commodity hardware. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 308–318, New York, NY, USA, 2008. ACM.

[64] NVIDIA Corporation. Tegra, 2015.

[65] S.-W. Olle, L. S´ebastien, and L. Johan. Evaluation of the energy efficiency of arm based processors for cloud infrastructure. *Turku Centre for Computer Science*, 2010.

[66] W. Orme. Debug and trace for multicore socs. *ARM white paper*, 2008.

[67] H. Ozdoganoglu, T. Vijaykumar, C. E. Brodley, B. A. Kuperman, and A. Jalote. Smashguard: A hardware solution to prevent security attacks on the function return address. *Computers, IEEE Transactions on*, 55(10):1271–1285, 2006.

[68] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent rop exploit mitigation using indirect branch tracing. In *USENIX Security*, pages 447–462, 2013.

[69] N. L. Petroni Jr, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot-a coprocessor-based kernel runtime integrity monitor. In *USENIX Security Symposium*, pages 179–194. San Diego, USA, 2004.

[70] N. L. Petroni Jr and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 103–115, 2007.

[71] M. Portela-García et al. On the use of embedded debug features for permanent and transient fault resilience in microprocessors. *Microprocessors and Microsystems*, 36(5):334–343, 2012.

[72] M. Portela-García, M. Grosso, M. Gallardo-Campos, M. S. Reorda, L. Entrena, M. García-Valderas, and C. López-Ongil. On the use of embedded debug features for permanent and transient fault resilience in microprocessors. *Microprocessors and Microsystems*, 36(5):334–343, 2012.

[73] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pages 135–148. IEEE, 2006.

[74] Qualcomm Technologies, Inc. Snapdragon, 2015.

[75] M. Rajagopalan, M. A. Hiltunen, T. Jim, and R. D. Schlichting. System call monitoring using authenticated system calls. *Dependable and Secure Computing, IEEE Transactions on*, 3(3):216–229, 2006.

[76] L. Samsung Electronics co. Exynos 4, 2012.

[77] L. Samsung Electronics co. Exynos 5 Dual, 2012.

[78] Samsung Electronics co., LTD. Exynos, 2015.

[79] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997.

[80] P. Saxena, R. Sekar, and V. Puranik. Efficient fine-grained binary instrumentationwith applications to taint-tracking. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 74–83. ACM, 2008.

[81] J. Seward. Origin tracking tool, valgrind release-3.4. 0. pre-release at svn co svn, 2014.

[82] J. Seward and N. Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference, General Track*, pages 17–30, 2005.

[83] K. Shahzad, A. Khalid, Z. E. Rákossy, G. Paul, and A. Chattopadhyay. Coarx: a coprocessor for arx-based cryptographic algorithms. In *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, pages 1–10. IEEE, 2013.

[84] K. Shankar and R. Lysecky. Non-intrusive dynamic application profiling for multitasked applications. In *Proceedings of the 46th Annual Design Automation Conference*, pages 130–135. ACM, 2009.

[85] T. shell strom linux shellcode repository, 2014.

[86] E. SOGETI. R&d lab,". *Analysis of the jailbreakme v3 font exploit*, 2012.

[87] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *ACM SIGOPS Operating Systems Review*, volume 38, pages 85–96. ACM, 2004.

[88] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *Secury and Privacy (SP), 2013 IEEE Symposium on*, pages 48–62, June 2013.

[89] P. Team. Address Space Layout Randomization, 2003.

[90] M. Tiwari, B. Agrawal, S. Mysore, J. Valamehr, and T. Sherwood. A small cache of large ranges: Hardware methods for efficiently searching, storing, and updating big dataflow tags. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 94–105, Washington, DC, USA, 2008. IEEE Computer Society.

[91] M. Tiwari, S. Mysore, and T. Sherwood. Quantifying the potential of program analysis peripherals. In *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*, pages 53–63. IEEE, 2009.

[92] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. Flexitaint: A programmable accelerator for dynamic taint propagation. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pages 173–184. IEEE, 2008.

[93] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic. Memtracker: Efficient and programmable support for memory access monitoring and debugging. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 273–284. IEEE, 2007.

[94] S. Vogl, R. Gawlik, B. Garmany, T. Kittel, J. Pfoh, C. Eckert, and T. Holz. Dynamic hooks: hiding control flow changes within non-control data. In *Proceedings of the 23rd USENIX conference on Security Symposium*, pages 813–828, 2014.

[95] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *NDSS*, 2007.

[96] E. Witchel, J. Cates, and K. Asanović. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, pages 304–316, New York, NY, USA, 2002. ACM.

[97] Y. Xia, Y. Liu, H. Chen, and B. Zang. Cfimon: Detecting violation of control flow integrity using performance counters. In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, pages 1–12. IEEE, 2012.

[98] I. Xilinx. Microblaze processor reference guide v13. 4. *reference manual*, 2011.

[99] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 116–127. ACM, 2007.

[100] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 559–573. IEEE, 2013.

[101] M. Zhang and R. Sekar. Control flow integrity for cots binaries. In *USENIX Security*, pages 337–352, 2013.

[102] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iwatcher: Efficient architectural support for software debugging. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, pages 224–235. IEEE, 2004.

[103] P. Zhou, R. Teodorescu, and Y. Zhou. Hard: Hardware-assisted lockset-based race detection. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 121–132. IEEE, 2007.

[104] Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall. *Privacy Scope: A precise information flow tracking system for finding application leaks.* PhD thesis, University of California, Berkeley, 2009.

초록

보안 모니터링 (security monitoring)이란 컴퓨터 상에서 프로그램이 실행할 때 발생되는 여러 행위들을 감시하여 공격의 존재를 발견하는 것이다. 최근 들어, 보안 위협이 증가함에 따라, 공격에 대한 탐지를 목표로 하는 이러한 보안 모니터링에 대한 연구가 많이 이루어져 왔다. 그 중 소프트웨어를 기반으로 한 연구들은 기존의 컴퓨터 시스템에 적용이 용이하다는 장점이 있으나, 큰 성능 오버헤드를 일으킨다는 문제가 있다. 반면, 하드웨어를 기반으로 한 솔루션들은 모니터링 성능을 크게 향상시켰지만, 이를 적용하기 위해 프로세서 아키텍쳐를 재설계해야 한다는 부담이 있다. 최근 들어 제안된 연구들에서는 이러한 아키텍쳐 상의 변화를 최소화하기 위하여, 보안 모니터링을 전담하는 전용 외부 하드웨어 모듈을 제안하기도 하였다. 하지만, 이러한 연구들 역시 호스트와 외부 하드웨어 사이의 통신에 따른 성능 오버헤드가 크다는 단점이 있었다.

이 학위 논문에서는 외부 하드웨어 엔진을 이용하는, 효율적인 보안 모니터링을 위한 연구들을 소개한다. 이러한 엔진들을 설계하는 데 있어 최우선 목표는 호스트 프로세서 내부 구조에 대한 수정을 전혀 요구하지 않는 것이다. 따라서, 이 학위 논문에서 소개되는 모니터링 하드웨어 엔진들은 호스트 프로세서와 연결될 시에, 기존에 존재하는 인터페이스들만을 사용할 수 있다. 이러한 룰을 지켜가면서, 본 논문에서는 다양한 아키텍쳐 디자인 공간을 탐색하여, 크게 3가지의 연구들을 소개한다. 시스템 버스만을 이용하는 하드웨어 엔진에서부터 출발하여, 이 논문의 마지막에서는 사용 프로세서들의 디버그 인터페이스를 사용하는 하드웨어 엔진들을 소개할 것이다. 이러한 디자인 공간 탐색을 통해, 이 학위 논문은 상용 플랫폼에서 보안 모니터링 하드웨어를 설계하는 데 있어 다양한 선택지를 제공해줄 수 있을 것이다.

주요어: 보안 모니터링, 디버그 인터페이스, SoC 수준 집적, 외부 하드웨어 엔진

학번: 2009-20920